The Report Committee for Andrew Richard Temple
Certifies that this is the approved version of the following report:

Software Optimization for Power Consumption in DSP Embedded Systems

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Christine Julien

Michael Brogioli

Software Optimization for Power Consumption in DSP Embedded Systems

by

Andrew Richard Temple, B.S.E.E

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2012

Dedication

This report is dedicated to my family and friends, who constantly inspire me.

Acknowledgements

It is thanks to the guidance and ideas from Michael Kardonik, Michael Brogioli, and Robert Oshana that this paper came about. I would also like to thank Dr. Christine Julien whose teachings inspired thought processes used in this report.

Abstract

Software Optimization for Power Consumption in DSP Embedded Systems

Andrew Richard Temple, M.S.E

The University of Texas at Austin, 2012

Supervisor: Christine Julien

This paper is intended to be a resource for programmers needing to optimize a DSP's power consumption strictly through software. The paper will provide a basic introduction into power consumption background, measurement techniques, and then go into the details of power optimization, focusing on three main areas: algorithmic optimization, taking advantage of hardware features (low power modes, clock control, and voltage control), and data flow optimization with a discussion into the functionality and power considerations when using fast SRAM type memories (common for cache) and DDR SDRAM. This work includes examples and results as tested on Freescale's current state of the art Digital Signal Processors.

Table of Contents

List of Tables

List of Figures

INTRODUCTION

This report will be reprinted as a textbook chapter in “Expert Guide: DSP for Embedded and Real-Time Systems,” published by Elsevier Inc., 2012.

One of the most important considerations in the product lifecycle of a DSP project is to understand and optimize the power consumption of the device. Power consumption is highly visible for handheld devices which require battery power to be able to guarantee certain minimum usage / idle times between recharging. The other main DSP applications: medical equipment, test, measurement, media, and wireless base station, are very sensitive to power as well - due to the need to manage heat dissipation [1] of increasingly powerful processors, power supply cost, and energy consumption cost, so the fact is that power consumption cannot be overlooked.

The responsibility of setting and keeping power requirements often falls on the shoulders of hardware designers, but the software programmer has the ability to provide a large contribution to power optimization. Often, the impact that the software engineer has to influence the power consumption of a device is overlooked or underestimated, as Oshana notes in the introduction to Power in [1].

The goal of this work is to discuss how software can be used to optimize power consumption, starting with the basics of what power consumption consists of, how to properly measure power consumption, and then moving on to techniques for minimizing power consumption in software at the algorithmic level, hardware level, and data flow. This will include demonstrations of the various techniques and explanations using Freescale StarCore DSPs of both how and why certain methods are effective at reducing power so the reader to take and apply this work to their application right away.

UNDERSTANDING POWER CONSUMPTION

Basics of Power Consumption

In general, when power consumption is discussed, the four main factors discussed for a device are the application, the frequency, the voltage and the process technology, so we need to understand why exactly it is that these factors are so important.

The application is highly important, so much so that the power profile for two handheld devices could differ to the point of making power optimization strategies the complete opposite. While we will be explaining more about power optimization strategy later on, the basic idea is clear enough to introduce in this section.

Take for example a portable media player vs. a cellular phone. The portable media player needs to be able to run at 100% usage for a long period of time to display video (full length movies), audio, etc. We will discuss this later, but the general power consumption profile for this sort of device would have to focus on algorithmic and data flow power optimization more than on efficient usage of low power modes.

Compare this to the cellular phone, which spends most of its time in an idle state, and during call time, the user is only talking a relatively small % of the time. For this small percentage of time, the processor may be heavily loaded performing encode/decode of voice and transmit/receive data. For the remainder of the call time, the phone is not so heavily tasked; performing procedures such as sending heartbeat packets to the cellular network and providing “comfort noise” to the user to let the user know the phone is still connected during silence. For this sort of a profile, power optimization would be focused first around maximizing processor sleep states to save as much power as possible, and then on data flow / algorithmic approaches.

In the case of process technology, the current cutting edge DSPs are based on 45nm technology, a decrease in size from its predecessor, the 65nm technology. What this smaller process technology provides is a smaller transistor. Smaller transistors consume less power and produce less heat, so are clearly advantageous to their predecessors.

Smaller process technology also generally enables higher clock frequencies, which is clearly a plus, providing more processing capability, but higher frequency, along with higher voltage, come at the cost of higher power draw. Voltage is the most obvious of these, as we learned in physics (and EE101), power is the product of voltage times current. So if a device requires a large voltage supply, power consumption increase is a fact of life.

While staying on our subject of P=V*I, the frequency is also directly part of this equation because current is a direct result of the clock rate. Another thing we learned in physics and EE101: when voltage is applied across a capacitor, current will flow from the voltage source to the capacitor until the capacitor has reached an equivalent potential. While this is an over-simplification, we can imagine that the clock network in a DSP consumes power in such a fashion. Thus at every clock edge, when the potential changes, current flows through the device until it reaches the next steady state. The faster the clock is switching, the more current is flowing, therefore faster clocking implies more power consumed by the DSP. Depending on the device, the clock circuit is responsible for consuming between 50% and 90% of dynamic device power, so controlling clocks is a theme that will be covered very heavily here.

Section 1.1: Static versus Dynamic Power Consumption

Total power consumption consists of two types of power: dynamic and static (also known as static leakage) consumption, so total device power is calculated as:

$$P_{total} = P_{Dynamic} + P_{Static}$$

As we have just discussed, clock transitions are a large portion of the dynamic consumption, but what is this "dynamic consumption"? Basically, in software we have control over dynamic consumption, but we do not have control over static consumption.

STATIC POWER CONSUMPTION

Leakage consumption is the power that a device consumes independent of any activity or task the DSP is running, because even in a steady state, there is a low "leakage" current path (via transistor tunneling current, reverse diode leakage, etc) from the device's V_{in} to ground. The only factors that affect the leakage consumption are: supply voltage, temperature, and process.

We have already discussed voltage and process in the introduction. In terms of temperature, it is fairly intuitive to understand why heat increases leakage current. Heat increases the mobility of electron carriers, which will lead to an increase in electron flow, causing greater static power consumption. As the focus of this text is software, this will be the end of static power consumption theory. Further details on temperature and carrier mobility can be found in [2].

DYNAMIC POWER CONSUMPTION

The dynamic consumption of the DSP includes the power consumed by the device actively using the cores, core subsystems, peripherals such as DMA, I/O (radio, Ethernet, PCIe, CMOS Camera), memories, and PLLs and clocks. At the low level, this can be translated to dynamic power is the power consumed by switching transistors, which are charging and discharging capacitances.

Dynamic power increases as we use more elements of the system, more cores, more arithmetic units, more memories, higher clock rates, or anything that could possibly increase the amount of transistors switching, or the speed at which they are switching. The dynamic consumption is independent of temperature, but still depends on voltage supply levels.

Section 1.2: Maximum, Average, Worst Case, and Typical Power

When measuring power, or determining power usage for a system, there are four main types of power that need to be considered: maximum power, average power, worst case power consumption, and typical power consumption.

Maximum and average power are general terms, used to describe the power measurement itself more than the effect of software or other variables on a device's power consumption.

Simply stated, maximum power is the highest instantaneous power reading measured over a period of time. This sort of measurement is useful to show the amount of decoupling capacitance required by a device to maintain a decent level of signal integrity (required for reliable operation).

Average power is intuitive at this point: technically the amount of energy consumed in a time period, divided by that time (power readings averaged over time). Engineers do this by calculating the average current consumed over time and use that to find power. Average power readings are what we are focusing on optimizing as this is the determining factor for how much power a battery or power supply must be able to provide for a DSP to perform an application over time, and this also used to understand the heat profile of the device.

Both worst case and typical power numbers are based on average power measurement. Worst case power, or the worst case power profile, describes the amount of average power a device will consume at 100% usage over a given period time. 100% usage infers to the processer utilizing the maximum number of available processing units (data and address generation blocks in the core, accelerators, bit masking, etc), memories, and peripherals simultaneously. This may be simulated by putting the cores put in an infinite loop of performing 6 or more instructions per cycle (depending on the available processing units in the core) while having multiple DMA channels continuously reading and writing from memory, and peripherals constantly sending and receiving data. Worst case power numbers are used by the system architect or board designer in order to provide adequate power supply to guarantee functionality under all worst case conditions.

In a real system, a device will rarely if ever draw the worst case power, as applications are not using all the processing elements, memory, and I/O for long periods of time, if at all. In general, a device provides many different I/O peripherals, though only a portion of them are needed, and the device cores may only need to perform heavy computation for small portions of time, accessing just a portion of memory. Typical power consumption then may be based off the assumed “general use case” example application that may use anywhere from 50 to 70% of the processors available hardware

components at a time. This is a major aspect of software applications that we are going to be taking advantage of in order to optimize power consumption.

In this section we have explained the differences of static vs. dynamic power, maximum vs. average power, process effect on power, and core and processing power effect on power. Now that the basics of what makes power consumption are covered, we will discuss power consumption measurement before going into detail about power optimization techniques.

MEASURING POWER CONSUMPTION

Now that background, theory, and vocabulary have been covered, we will move on to taking power measurements. We will discuss the types of measurements used to get different types of power readings (such as reading static vs. dynamic power), and use these methods in order to test optimization methods used later in the text.

Measuring power is hardware dependent: some DSPs provide internal measurement capabilities, DSP manufacturers also may provide "power calculators" which give some power information, there are a number of power supply controller ICs which provide different forms of power measurement capabilities, some power supply controllers called VRMs ("Voltage Regulator Modules") have these capabilities internal to them to be read over peripheral interfaces, and finally, the old fashioned method of connecting an ammeter in series to the DSP's power supply.

MEASURING POWER USING AN AMMETER

The "old fashioned" method is to measure power is via the use of an external power supply connected in series to the positive terminal of an ammeter, which connects via the negative connector to the DSP device power input, as shown in Figure 1 below.

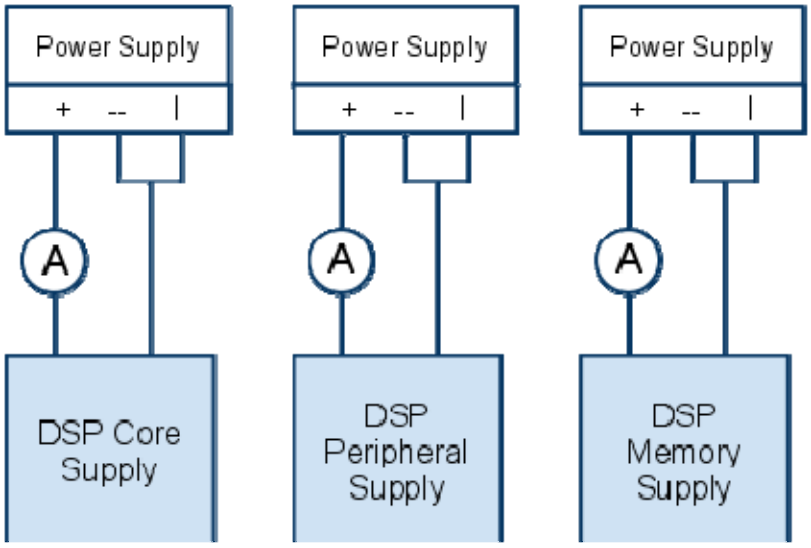


Figure 1: Measuring Power via Ammeters

Note that there are three different setups shown in Figure 1, which are all for a single DSP. This is due to the fact that DSP power input is isolated, generally between cores, (possibly multiple supplies) peripherals, and memories. This is done by design in hardware as different components of a device have different voltage requirements, and this is useful to use to isolate (and eventually optimize) the power profile of individual components.

In order to properly measure power consumption, the power to each component must be properly isolated, which in some cases may require board modification, specific jumper settings, etc. The most ideal situation is to be able to connect the external supply/ammeter combo as close as possible to the DSPs power input pins.

Alternatively, one may measure the voltage drop across a (shunt) resister which is in series with the power supply and a DSP's power pins. By measuring the voltage drop across the resistor, current is found simply by calculating $I = V / R$.

MEASURING POWER USING A HALL SENSOR TYPE IC

In order to simplify efficient power measurement, many DSP vendors are building boards that use a Hall-Effect based sensor. When Hall sensors are placed on a board in the current path to the device's power supply, it generates a voltage equivalent to the current times some coefficient with an offset. In the case of Freescale's MSC8144 DSP Application Development System board, an Allegro ACS0704 Hall Sensor is provided on the board which enables such measurement. With this board, the user can simply place a scope to the board, and view the voltage signal over time, and use this to calculate average power using Allegro's current to voltage graph, shown in Figure 2.

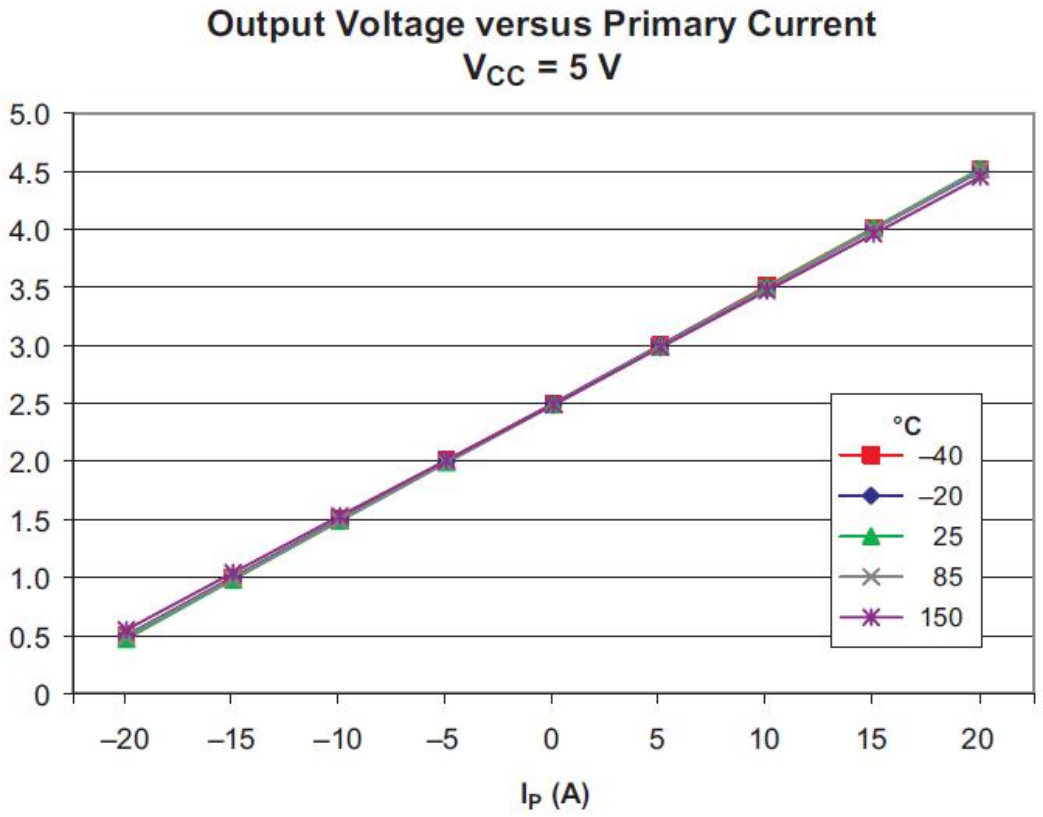


Figure 2: Hall Effect IC Voltage to Current Graph [3]

Using Figure 2, we can calculate input current to a device based on measuring potential across Vout as:

$$I = (Vout - 2.5) * 10A$$

VRMs ("Voltage Regulator Module" Power Supply ICs)

Finally, some VRMs (power supply controller ICs), which are used to split a large input voltage into a number of smaller ones to supply individual sources at varying potentials, measure current/power consumption and store the values in registers to be read by the user. Measuring current via the VRM requires no equipment, but this sometimes comes at the cost of accuracy and real time measurement. For example, the PowerOne ZM7100 series VRM (also used on the MSC8144ADS) provides current readings for each supply, but the current readings are updated once every 0.5 to 1 seconds, and the reading accuracy is on the order of ~20%, so instantaneous reading for maximum power is not possible, and fine tuning and optimization may not be possible using such devices.

In addition to deciding a specific method for measuring power in general, different methods exist to measure dynamic power versus static leakage consumption. The static leakage consumption data is useful in order to have a floor for our low power expectations, and to understand how much power the actual application is pulling vs. what the device will pull in idle. We can then subtract that from the total power consumption we measure in order to determine the dynamic consumption the DSP is pulling, and work to minimize that.

Section 2.1: Static Power Measurement:

Leakage consumption on the DSP can usually be measured while the device is placed in a low power mode, assuming that the mode shuts down clocks to all of the DSP core subsystems and peripherals. In the case that the clocks are not shut down in low power mode, the PLLs should be bypassed, and then the input clock, should be shut down, thus shutting down all clocks and eliminating clock and PLL power consumption from the static leakage measurement.

Additionally, static leakage should be measured at varying temperatures since leakage varies based on temperature. Creating a set of static measurements based on temperature (and voltage) provides valuable reference points for determining how much dynamic power an application is actually consuming at these temperature/voltage points.

Section 2.2: Dynamic Power Measurement

The power measurements should separate the contribution of each major module in the device to give the engineer information about what effect a specific configuration will have on a system's power consumption. As noted above, dynamic power is found simply by measuring the total power (at a given temperature) and then subtracting the leakage consumption for that given temperature using the initial static measurements from above.

Initial dynamic measurement tests include running sleep state tests, debug state tests, and a NOP test. Sleep state and debug state tests will give the user insight into the cost of enabling certain clocks in the system. A NOP test, as in a loop of NOP commands, will provide a baseline dynamic reading for your core's consumption when mainly using the fetch unit of the device, but no arithmetic units, address generation, bit mask, memory management, etc.

When comparing specific software power optimization techniques, we compare the before and after power consumption numbers of each technique in order to determine the effect of that technique.

PROFILING YOUR APPLICATIONS POWER CONSUMPTION

Before optimizing an application for power, the programmer should get a baseline power reading of the section of code being optimized. This provides a reference point for measuring optimizations, and also ensures that the alterations to code are in fact decreasing total power, and not the opposite. In order to do this, the programmer needs to generate a sample power test which acts as a snapshot of the code segment being tested.

This power test case generation can be done by profiling code performance using a high end profiler to gain some base understanding of the % of processing elements and memory used. We can demonstrate this in Freescale's CodeWarrior for StarCore IDE, by creating a new example project using the CodeWarrior stationary with the profiler enabled, then compiling, and running the project. The application will run from start to finish, at which point the user may select a profiler view and get any number of statistics.

Using relevant data such as the % of ALU's used, AGU's used, code hot-spots, and knowledge of memories being accessed, we can get a general idea of where our code will spend the most time (and consume the most power). We can use this to generate a basic performance test which runs in an infinite loop, enabling us to profile the average "typical" power of an important code segment.

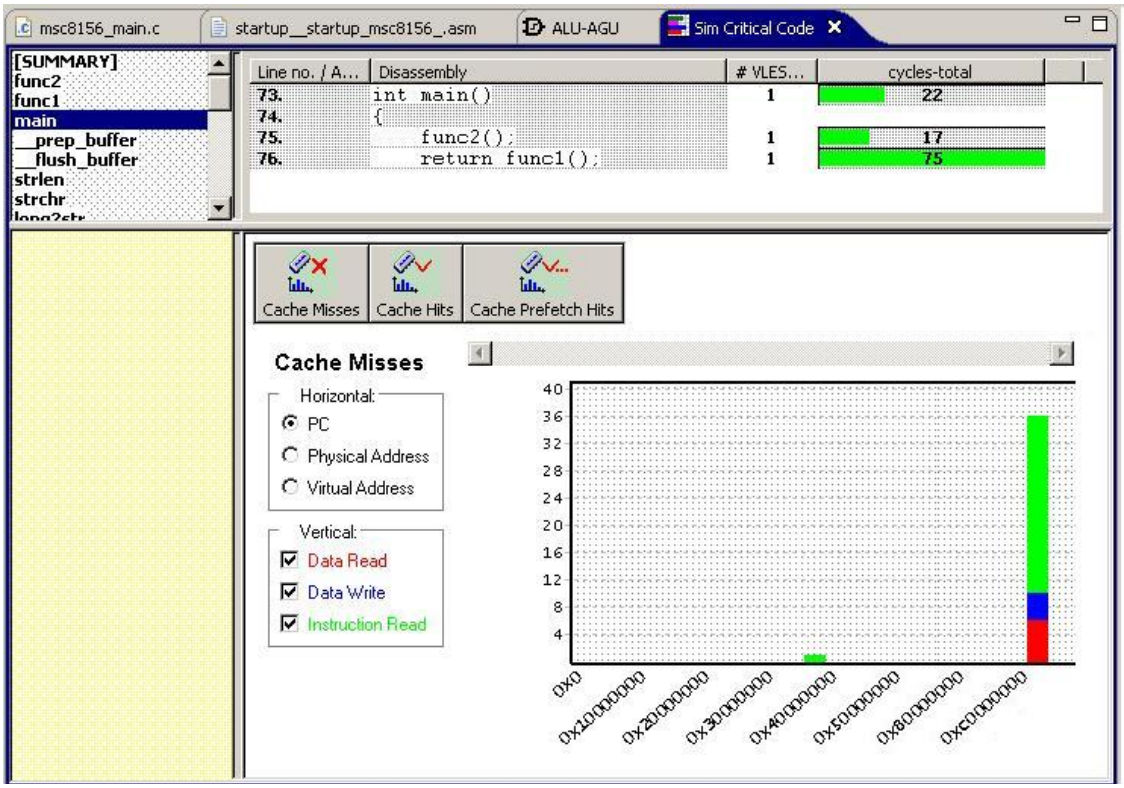
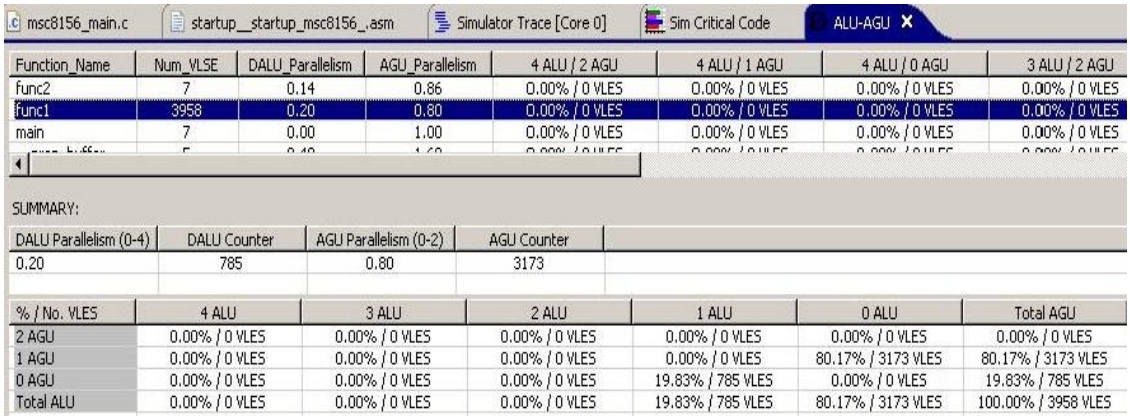


Figure 3: Profiling for hot spots

In the standard Freescale CodeWarrior example project, there are 2 main functions: func1 and func2. Profiling the example code, we can see from the Figure 3 that the vast majority of cycles are consumed by the func1 routine. This routine is located in M2 memory and is reading data from cacheable M3 memory (meaning possible causing write back accesses to L2 and L1 cache). By using the profiler (as per Figure 4 below), information regarding the % ALU, % AGU can be extracted. This enables us to understand the actual DSP core usage this way. In the case of the SC3850 core in the MSC8156 DSP, there are 2 AGUs and 4 DALUs which enable the programmer to run up to 6 instructions in parallel. Each AGU (address generation unit) enables the core to perform effective address calculations needed for accessing data operands in memory. Each DALU (dual arithmetic logic unit) enables arithmetic and can perform two

multiplications (or multiply accumulates aka MACs) per cycle. The goal of the programmer seeking to get the most out of the core for each cycle would be to maximize the use of all 6 of these core elements per cycle, ideally performing 2 AGU operations and 4 DALU operations (8 MACs) per cycle.

We can effectively simulate this by turning the code into an infinite loop, adjusting the I/O, and compiling at the same optimization level, and verifying that we see the same performance breakdown.


msc8156_main.c | startup__startup_msc8156_.asm | Simulator Trace [Core 0] | Sim Critical Code | ALU-AGU

Function_Name	Num_VLSE	DALU_Parallelism	AGU_Parallelism	4 ALU / 2 AGU	4 ALU / 1 AGU	4 ALU / 0 AGU	3 ALU / 2 AGU
func2	7	0.14	0.86	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES
func1	3958	0.20	0.80	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES
main	7	0.00	1.00	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES

SUMMARY:

DALU Parallelism (0-4)	DALU Counter	AGU Parallelism (0-2)	AGU Counter
0.20	785	0.80	3173

% / No. VLES	4 ALU	3 ALU	2 ALU	1 ALU	0 ALU	Total AGU
2 AGU	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES
1 AGU	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	80.17% / 3173 VLES	80.17% / 3173 VLES
0 AGU	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	19.83% / 785 VLES	0.00% / 0 VLES	19.83% / 785 VLES
Total ALU	0.00% / 0 VLES	0.00% / 0 VLES	0.00% / 0 VLES	19.83% / 785 VLES	80.17% / 3173 VLES	100.00% / 3958 VLES

Figure 4: Core Component (%ALU %AGU) Utilization

We can then set a break point, re-run our application, and confirm that the device usage profile is in line with our original code. If not, we can adjust compiler optimization level or our code until it matches the original application.

This method is quick and effective for measuring power consumption for various loads. By having the infinite loop, testing is much easier as we are simply comparing steady state current readings of optimized and non-optimized code in hopes of getting lower numbers. We can use this to measure numerous metrics such as average power over time, average power per instruction, average power per cycle, and energy (power * time) in joules for some time t. For measuring specific algorithms and power saving

techniques, we will form small routines using similar methods and then optimize measure the power savings over time.

This section has explained a few different methods for measuring static power, dynamic power, and how to profile power for an application. It also covered the availability of power calculators from DSP manufacturers, which sometimes may quicken the power estimation process. Using these tools will enable effectively measuring and confirming the knowledge shared in the next section of this text, which covers the software techniques for optimizing power consumption.

MINIMIZING POWER CONSUMPTION

There are three main types of power optimization covered in this text: hardware supported features, data path optimization, and algorithmic optimization. Algorithmic optimization refers to making changes in code to affect how the DSP's cores process data, such as how instructions or loops are handled, whereas hardware optimization, as discussed here, focuses more on how to optimize clock control and power features provided in hardware. Data flow optimization focuses on working to minimize the power cost of utilizing different memories, buses, and peripherals where data can be stored or transmitted by taking advantages of relevant features and concepts.

Section 3.1: Hardware Support:

LOW POWER MODES (INTRODUCTION TO DEVICES)

DSP applications normally work on tasks in packets, frames, or chunks. For example, in a media player, frames of video data may be coming in at 60 frames per second to be decoded, while the actual decoding work may take the processor orders of magnitude less than 1/60 of a second, giving us a chance to utilize sleep modes, shut down peripherals, and organize memory all to reduce power consumption and maximize efficiency.

We must also keep in mind that power consumption profile varies based on application. For instance, 2 differing hand held devices: an mp3 player and a cellular phone, will have two very different power profiles.

The cellular phone spends most of its time in an idle state, and when in a call, is still not working at full capacity during the entire call duration as speech will commonly contain pauses which are long in terms of the DSP processor's clock cycles.

For both of these power profiles, software enabled low power modes (modes/features/controls) are used to save power, and the question for the programmer is how to use them efficiently. The most common modes available consist of power gating, clock gating, voltage scaling, and clock scaling [4].

POWER GATING

Power Gating uses a current switch to cut off a circuit from its power supply rails during standby mode, to eliminate static leakage when the circuit is not in use. Using power gating leads to a loss of state and data for a circuit, meaning that using this requires storing necessary context/state data to active memory. As DSPs are moving more and more towards being full SoC solutions with many peripherals, some peripherals may be unnecessary for certain applications. Power gating may be available to completely shut off such unused peripherals in a system, and the power savings attained from power gating depends on the specific peripheral on the specific device in question.

It is important to note that in some cases, documentation will refer to powering down a peripheral via clock gating, which is different from power gating. It may be possible to gate a peripheral by connecting the power supply of a certain block to ground, depending on device requirements and interdependency on a power supply line. This is possible in software in certain situations, such as when board/system level power is controlled by an on-board IC (such as the PowerOne IC), which can be programmed and updated via an I2C bus interface. As an example, the MSC8156 DSP has this option for the MAPLE DSP baseband accelerator peripheral and a portion of M3 memory.

CLOCK GATING

Clock gating, as the name implies, shuts down clocks to a circuit or portion of a clock tree in a device. As dynamic power is consumed during state change triggered by clock toggling (as we discussed in the introductory portion of this text), clock gating enables the programmer to cut dynamic power through the use of a single (or a few) instructions. Clocking of a DSP is generally separated into trees stemming from a main clock PLL into various clock domains as required by design for core, memories, and peripherals, and DSPs generally enable levels of clock gating in order to customize a power savings solution.

Freescale's MSC815x Low Power Modes:

Freescale DSPs provide various levels of clock gating in the core subsystem and peripheral areas. Gating clocks to a core may be done in the form of STOP and WAIT instructions. STOP mode gates clocks to the DSP core and the entire core subsystem (L1 and L2 Caches, M2 memory, memory management, debug and profile unit) aside from internal logic used for waking from STOP state.

In order to safely enter STOP mode, as one may imagine, care must be taken to ensure accesses to memory and cache are all complete, and no fetches/prefetches are underway.

The recommended process is:

1. Terminate any open L2 prefetch activity
2. Stop all internal and external accesses to M2/L2 memory
3. Close the subsystem slave port window (peripheral access path to M2 memory) by writing to the core subsystem slave port general configuration register

4. Verify slave port is closed by reading the register, and also testing access to the slave port (at this point, any access to the core's slave port will generate an interrupt).
5. Ensure STOP ACK bit is asserted in General Status Register to show subsystem is in stop state
6. Enter Stop mode

STOP state can be exited by initiating an interrupt. There are other ways to exit from STOP state, including a reset or debug assertion from external signals.

The WAIT state gates clocks to the core and some of the core subsystem aside from the interrupt controller, debug and profile unit,timer, and M2 memory, which enables faster entering and exiting from WAIT state, but at the cost of greater power consumption. To enter wait state, the programmer may simply use the WAIT instruction for a core. Exiting WAIT, like STOP, may also be done via an interrupt.

A particularly nice feature of these low power states on the Freescale DSPs is that both STOP and WAIT mode can be exited via either an enabled or disabled interrupt. Wake up via an enabled interrupt follows standard interrupt handling procedure: the core takes the interrupt, does a full context switch, and then the program counter jumps to the interrupt service routine before returning to the instruction following the segment of code that executed WAIT (or STOP) instruction. This requires a comparatively large cycle overhead, which is where disabled interrupt waking becomes quite convenient. When using a disabled interrupt to exit from either WAIT or STOP state, the interrupt signals the core using an interrupt priority that is not "enabled" in terms of the core's global interrupt priority level (IPL), and when the core wakes, it resumes execution where it left

off without executing a context switch or any ISR. An example using a disabled interrupt for waking the MSC8156 is provided at the end of this section.

Clock gating to peripherals is also enabled, where the user may gate specific peripherals individually as needed. This is available for the MSC8156's serial interface, Ethernet controller (QE), DSP accelerators (MAPLE), and DDR. As with STOP mode, when gating clocks to any of these interfaces, the programmer must ensure that all accesses are completed beforehand. Then, via the System Clock Control register, clocks to each of these peripherals may be gated. In order to come out of the clock gated modes, a Power on Reset is required, so this is not something that can be done and undone on the fly in a function, but rather a setting that is decided at system configuration time. .

Additionally, partial clock gating is possible on the High Speed Serial Interface components (SERDES, OCN DMA, SRIO, RMU, PCI Express), and DDR so that they may be temporarily put in a "doze state" in order to save power, but still maintain the functionality of providing an acknowledge to accesses (in order to prevent internal or external bus lockup when accessed by external logic).

Texas Instruments C6000 Low Power Modes

Another popular DSP family on the market is the C6000 series DSP from Texas Instruments (TI). TI DSPs in the C6000 family provide a few levels of clock gating, depending on the generation of C6000. For example, the previous generation C67x floating point DSP has low power modes called "power down modes". These modes include PD1, PD2, PD3, and "peripheral power down", each of which gates clocking to various components in the silicon.

For example, PD1 mode gates clocks to the C67x CPU (processor core, data registers, control registers, and everything else within the core aside from the interrupt

controller). The C67x can wake up from PD1 via an interrupt into the core. Entering PD1. Entering power down mode PD1 (or PD2 / PD3) for the C67x, is done via a register write (to CSR). The cost of entering PD1 state is ~9 clock cycles plus the cost of accessing the CSR register. As this power down state only affects the core (And not cache memories), it is not comparable to the Freescale's STOP or WAIT state.

The 2 deeper levels of power down, PD2 and PD3, effectively gates clocks to the entire device (all blocks which use an internal clock: internal peripherals, the CPU, cache, etc). The only way to wake up from PD2 and PD3 clock gating is via a reset, so PD2 and PD3 would not be very convenient or efficient to use mid-application.

The newer Keystone TI DSP family (C66x), which combine floating point and fixed point architectures from previous C6000 devices, retains the PD1, PD2, and PD3 states in the CSR register.

The C66xx provides the ability to gate a subset of the peripherals independently by clock domain, similar to the Freescale DSPs.

CLOCK AND VOLTAGE CONTROL

Some devices have the ability to scale voltage or clock, which may help optimize the power scheme of a device/application. Voltage scaling, as the name implies, is the process of lowering or raising the power. In the section on measuring current, VRMs were introduced as one method. The main purpose of a VRM (Voltage Regulator Module) is to control the power/voltage supply to a device. Using a VRM, voltage scaling may be done through monitoring and updating voltage ID (VID) parameters.

In general, as voltage is lowered, frequency / processor speed is sacrificed, so generally voltage would be lowered when demand of a DSP core or a certain peripheral is reduced.

The TI C6000 devices provide a flavor of voltage scaling called SmartReflex®. SmartReflex® enables automatic voltage scaling through a pin interface which provides VID to a Voltage Regulator Module (VRM). As the pin interface is internally managed, the software engineer does not have much affect over this, so we will not cover any programming examples for this.

Clock control is available in many DSPs, such as the MSC8144 from Freescale, which allows changing the values of various PLLs in runtime. In the case of the MSC8144, updating the internal PLLs requires relocking the PLLs, where some clocks in the system may be stopped, and this must be followed by a soft reset (reset of the internal cores). Because of this inherent latency, clock scaling is not very feasible during normal heavy operation, but may be considered if a DSP's requirements over a long period of time are reduced (such during times of low call volume during the night for DSPs on a wireless base station).

When considering clock scaling, we must keep the following in mind: During normal operation, running at a lower clock allows for lower dynamic power consumption, assuming clock and power gating are never used. In practice, running a processor at a higher frequency allows for more "free" cycles, which, as previously noted, can be used to hold the device in a low power / sleep mode - thus offsetting the benefits of such clock scaling.

Additionally, for the case of the MSC8144, updating the clock for custom cases is time intensive, and for many other DSPs, not an option at all - meaning clock frequency has to be decided at device reset/power on time, so the general rule of thumb is to enable enough clock cycles with some additional headroom for the real time application being run, and to utilize other power optimization techniques. Determining the amount of headroom varies from processor to processor and application to application - at which point it makes sense to profile your application in order to understand the amount of cycles required for a packet/frame, and the core utilization during this time period.

Once this is understood, measuring the power consumption for such a profile can be done, as demonstrated earlier in this text in the Profiling Power section. Measure the average power consumption at your main frequency options. (In MSC8144 and MSC815x, this could be 800MHz and 1GHz), and then average in idle power over the headroom slots in order to get a head to head comparison of the best case power consumption.

CONSIDERATIONS AND USAGE EXAMPLES OF LOW POWER MODES

Here we will summarize the main considerations for low power mode usage, and then close with a coding example demonstrating low power mode usage in a real time multimedia application.

1. Consider available block functionality when in low power mode:

   When in low power modes, we have to remember that certain peripherals will not be available to external peripherals, and peripheral buses may also be affected. As noted earlier in this section, devices may take care of this, but this is not always the case. If power gating a block, special care must be taken regarding shared external buses, clocks, and pins.

Additionally, memory states and validity of data must be considered. We will cover this when discussing cache and DDR in the next section

2. Consider the overhead of entering and exiting low power modes:

   When entering and exiting low power modes, in addition to overall power savings, the programmer must ensure the cycle overhead of actually entering and exiting the low power mode does not break real time constraints.

   Cycle overhead may also be affected by the potential difference in initiating a low power mode by register access as opposed to by direct core instructions.

LOW POWER EXAMPLE

To demonstrate low power usage, we will refer to the Motion JPEG (MJPEG) application. As a quick intro: The MJPEG demo is a real time Smart DSP OS demo intended to be run on an MSC8144 or MSC8156 development board.

With the MJPEG demo, raw image frames are sent from a PC to the DSP over Ethernet. Each Ethernet packet contains 1 block of an image frame. A full raw QVGA image uses ~396 blocks plus a header. The DSP encodes the image in real time (adjustable from 1 to 30+ frames per second), and sends the encoded Motion JPEG video back over Ethernet to be played on a demo GUI in the PC. The flow and a screenshot of this GUI are shown in the following figure.

The GUI will display not only the encoded JPEG image, but also the core utilization (as a percentage of the maximum core cycles available).

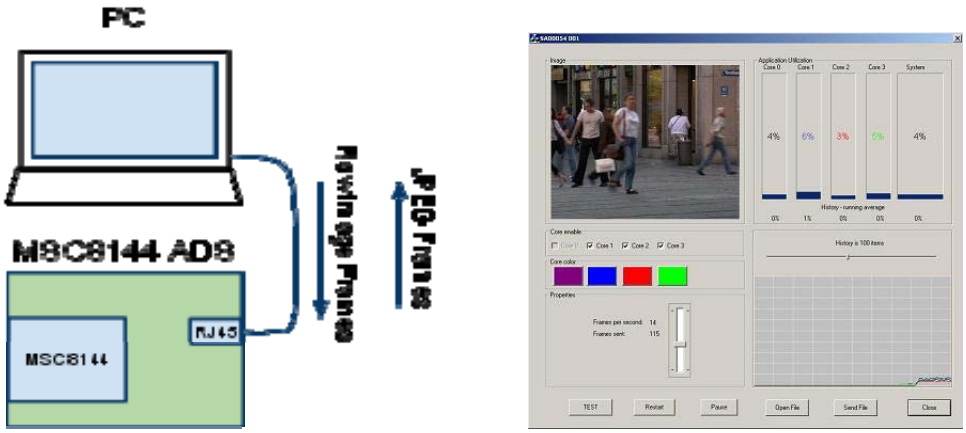


Figure 5: SmartDSP OS Motion JPEG Demo

For this application, we need to understand how many cycles encoding a frame of JPEG consumes. Using this we can determine the maximum frame rate we can use and, in parallel, also determine the maximum down time we have for low power mode usage. If we are close to the maximum core utilization for the real-time application, then using low power modes may not make sense (may break real-time constraints).

As noted in previous sections, we could simply profile the application to see how many cycles are actually spent per image frame, but this is already handled in the MJPEG demo's code using the core cycle counters in the OCE (On Chip Emulator). The OCE is the hardware block on the MSC81xx series DSPs that the profiler utilizes to get core cycle counts for use in code profiling.

The MJPEG code in this case counts the number of cycles a core spends doing actual work (handling an incoming Ethernet interrupt, dequeueing data, encoding a block of data into JPEG format, enqueueing/sending data back over Ethernet.

The # of core cycles required to process a single block encode of data (and supporting background data movement) is measured to be on the order of 13000 cycles. For a full JPEG image (~396 image blocks and Ethernet packets), this is approximately 5

million cycles. So 1 JPEG frame a second would work out to be 0.5% of a cores potential processing power considering a 1GHz core that is handling all Ethernet I/O, interrupt context switches, etc.

$$Cycles_{Buffer\ Mgmt\ \&\ Enetk} = 13{,}000$$

$$Cycles_{JPEG\ Frame} = Cycles_{Buffer\ Mgmt\ \&\ Enetk} \times 396 = 5{,}148{,}000$$

$$Core\ Utilization_{@1FPS}(\%) = 30\frac{[0] \times OCE\ Count}{1{,}000{,}000{,}000} = 15.4\%$$

As the MSC81xx series DSPs have up to 6 cores, and only 1 core would have to manage Ethernet I/O, in a full multicore system, utilization per core drops to a range of 3 to 7%. A master core acts as the manager of the system, managing both Ethernet I/O, intercore communication, and JPEG encoding, while the other slave cores are programmed to solely focus on encoding JPEG frames. Because of this intercore communication and management, the drop in cycle consumption from 1 core to 4 or 6 is not linear.

Based on cycle counts from the OCE, we can run a single core, which is put in a sleep state for 85% of the time, or a multicore system which uses sleep state up to 95% of the time.

This application also uses only a portion of the SoC peripherals (Ethernet, JTAG, a single DDR, and M3 memory). So we can save power by gating the full HSSI System (Serial Rapid IO, PCI Express), the MAPLE Accelerator, and the second DDR controller. Additionally, for our GUI demo, we are only showing 4 cores, so we can gate cores 4 and 5 without affecting this demo as well.

Based on the above, and what we have discussed in this section, here is the plan we want to follow:

At Application start up:

3. Clock Gate the unused MAPLE Accelerator Block (MAPLE described later in this work).

    NOTES

    1. MAPLE power pins share a power supply with core voltage. If the power supply to MAPLE was not shared, we could completely gate power. Due to shared pins on the development board, the most effective choice we have is to gate the MAPLE clock.
    2. MAPLE automatically goes into a doze state, which gates part of the clocks to the block, when it is not in use. Because of this, power savings from entirely gating MAPLE may not be massive.

4. Clock gate the unused HSSI (High Speed Serial Interface)

    NOTES

    1. We could also put MAPLE into a doze state, but this gates only part of the clocks. Since we will not be using any portion of these peripherals, complete clock gating is more power efficient.

5. Clock gate the unused second DDR controller

    NOTES

    1. When using VTB, SmartDSP OS places buffer space for VTB in the second DDR memory, so we need to be sure that this is not needed.

During Application Runtime

At runtime, QE (Ethernet Controller), DDR, and class CLASS, and cores 1-4 will be active. Things we must consider for these components include:

1. The Ethernet Controller cannot be shut down or put into a low power state - as this is the block that receives new packets (JPEG blocks) to encode. Interrupts from the Ethernet Controller can be used to wake our master core from low power mode.
2. Active core low power modes:
    a. WAIT mode enables core power savings, while allowing the core to be woken up in just a few cycles by using a disabled interrupt to signal exit from WAIT.
    b. STOP mode enables greater core savings by shutting down more of the subsystem than WAIT (including M2), but requires slightly more time to wake due to more hardware being re-enabled. If data is coming in at high rates, and the wake time is too long, we could get an overflow condition, where packets are lost. This is unlikely here due to the required data rate of the application.
3. The first DDR contains sections of program code and data, including parts of the Ethernet handling code. (This can be quickly checked and verified by looking at the program's .map file.) Because the Ethernet controller will be waking the master core from WAIT state, and the 1st thing the core will need to do out of this state is to run the Ethernet handler, we will not put DDR0 to sleep.

We can use the main background routine for the application to apply these changes without interfering with the RTOS. This code segment is shown below with power down related code in green:

```
static void appBackground(void)
{
    os_hwi_handle hwi_num;

    if (osGetCoreID() == 0)
    {
        *((unsigned int*)0xfff28014) = 0xF3FCFFFB;//HSSI CR1
```

```
            *((unsigned int*)0xfff28018) = 0x0000001F;//HSSI CR2
            *((unsigned int*)0xfff28034) = 0x20000E0E; //GCR5
            *((unsigned int*)0xfff24000) = 0x00001500; //SCCR
        }
        osMessageQueueHwiGet(CORE0_TO_OTHERS_MESSAGE, &hwi_num);
        while(1)
        {
            osHwiSwiftDisable();
            osHwiEnable(OS_HWI_PRIORITY10);
            stop();//wait();
            osHwiEnable(OS_HWI_PRIORITY4);
            osHwiSwiftEnable();
            osHwiPendingClear(hwi_num);
            MessageHandler(CORE0_TO_OTHERS_MESSAGE);

        }
    }
```

Note that the clock gating is must be done by only one core as these registers are system level and access is shared by all cores.

This code example demonstrates how a programmer using the SmartDSP OS can make use of the interrupt APIs in order to recover from STOP or wait state without actually requiring a context switch. In the MJPEG player, as noted above, raw image blocks are received via Ethernet (with interrupts), and then shared via shared queues (with interrupts). The master core will have to use context switching to read new Ethernet frames here, but slave cores only need to wake up and go to the MessageHandler function.

We take advantage of this fact by enabling only higher priority interrupts before going to sleep:

```
            osHwiSwiftDisable();
            osHwiEnable(OS_HWI_PRIORITY10);
```

Then when a slave core is asleep, if a new queue message arrives on an interrput, the core will be woken up (on context switch), and standard interrupt priority levels will be restored. The core will then go and manage the new message without context switch overhead by calling the MessageHandler() function.

In order to verify our power savings, we will take a baseline power reading before optimizing across the relevant power supplies, and then measure the incremental power savings of each step.

The MSC8156ADS board has power for cores, accelerators, HSSI, and M3 memory connected to the same power supply, simplifying data collection. Since these supplies and DDR are the only blocks we are optimizing, we shall measure improvement based on these supplies alone.

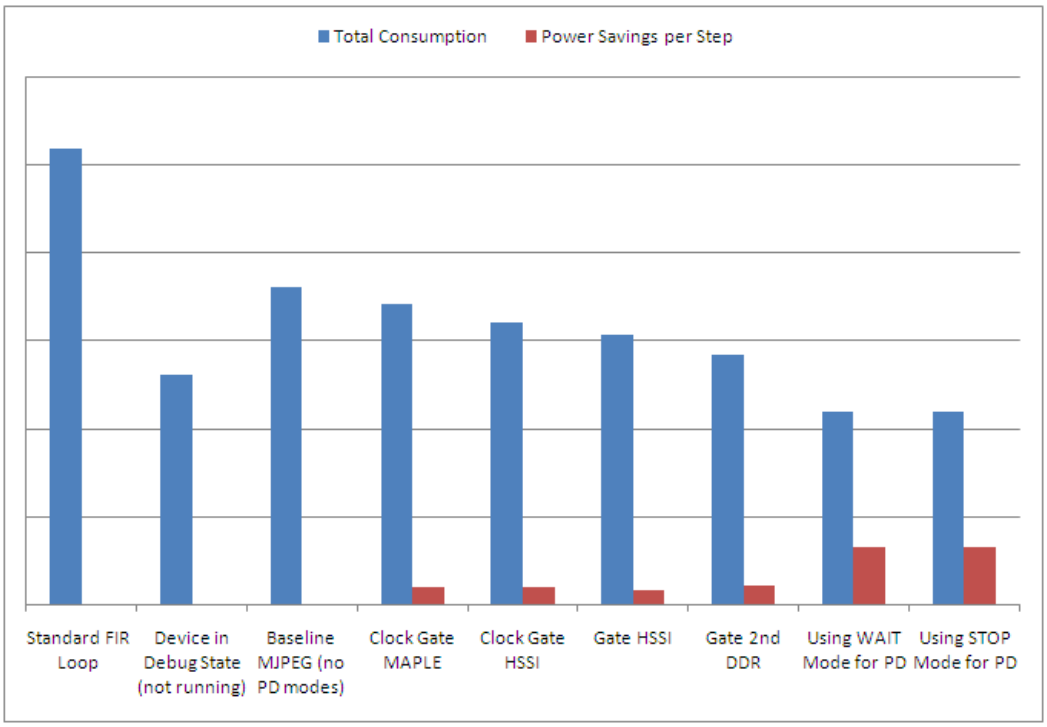


Figure 6: Power Consumption Savings in PD Modes

Figure 6 provides a visual on the relative power consumed by the relevant power supplies (1V: Core, M3, HSSI, MAPLE Accelerators, and DDR) across the power down

steps used above. Note that actual power numbers are not provided to avoid any potential non-disclosure issues.

The first two bars provide reference points - indicating the power consumption for these supplies using a standard FIR filter in a loop and the power consumption when the cores are held in debug state (not performing any instructions, but not in a low power mode). With our steps we can see that there was nearly a 50% reduction in power consumption across the relevant supplies for the Motion JPEG demo with the steps laid out above, with each step providing ~5% reduction in power with the exception of the STOP and WAIT power modes, which are closer to 15-20% savings.

One thing to keep in mind is that, while the MJPEG demo is the perfect example to demonstrate low power modes, it is not highly core intensive, so as we progress through different optimization techniques, we will be using other examples as appropriate.

Section 3.2: Optimizing Data Flow

Reducing Power Consumption for Memory Accesses

Due to clocks having to be activated not only in the core components, but also in buses, and memory cells, memory related functionality can be quite power hungry, but luckily, memory access and data paths can also be optimized to reduce power. This section will cover methods to optimize power consumption with regards to memory accesses to DDR and SRAM memories by utilizing knowledge of the hardware design of these memory types. Then we will cover ways to take advantage of other specific memory setups at the SoC level. Common practice is to optimize memory in order to maximize the locality of critical or heavily used data and code by placing as much in cache as possible. Cache misses incur not only core stall penalties, but also power penalties as more bus activity is needed, and higher level memories (internal device SRAM, or external device DDR) are activated and consume power. As a rule, access to higher level memory such as DDR are not as common as internal memory accesses, so high level memory accesses are easier to plan, and thus optimize.

DDR Overview

The highest level of memory we will discuss here is external DDR memory. To optimize DDR accesses in software, first we need to understand the hardware that the memory consists of. DDR SDRAM, as the DDR (dual data rate) name implies, takes advantage of both edges of the DDR clock source in order to send data, thus doubling the effective data rate at which data reads and writes may occur. DDR provides a number of different types of features which may affect total power utilization, such as EDC (error detection), ECC (error correction), different types of bursting, programmable data refresh

rates, programmable memory configuration allowing physical bank interleaving, page management across multiple chip selects, and DDR specific sleep modes.

Key DDR Vocabulary to be Discussed:

- **Chip Select** (also known as **Physical Bank**) - selects a set of memory chips (specified as a "rank") connected to the memory controller for accesses.
- **Rank** - specifies a set of chips on a DIMM to be accessed at once. A **Double Rank** DIMM, for example, would have two sets of chips - differentiated by chip select. When accessed together, each rank allows for a data access width of 64 bits (or 72 with ECC).
- **Rows** are address bits enabling access to a set of data, known as a "**page**" - so row and page may be used interchangeably.
- **Logical banks**, like row bits, enable access to a certain segment of memory. By standard practice, the row bits are the MSB address bits of DDR, followed by the bits to select a logical bank, finally followed by column bits.
- **Column** bits are the bits used to select and access a specific address for reading or writing

On a typical DSP, the DSPs' DDR SDRAM controller is connected to either discrete memory chips, or a DIMM (Dual Inline Memory Module), which contains multiple memory components (chips). Each discrete component/chip contains multiple logical banks, rows, and columns which provide access for reads and writes to memory. The basic idea of how a discrete DDR3 memory chip's layout is shown in figure 7 below.

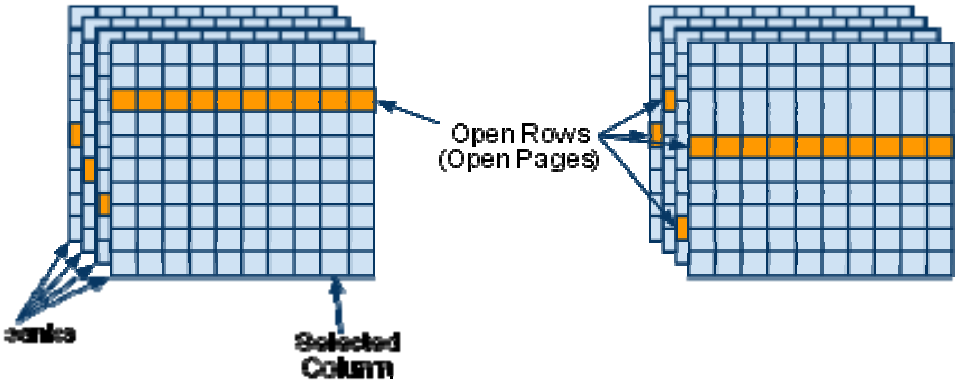


Figure 7: Basic Drawing of a Discrete DDR3 Memory Chip's Rows/Columns

Standard DDR3 discrete chips are commonly made up of 8 logical banks, which provide addressability as shown above. These banks are essentially tables of rows and columns. The action to select a row effectively opens that row (page) for the logical bank being addressed. So different rows can be simultaneously open in different logical banks, as illustrated by the active or open rows highlighted in the picture. A column selection gives access to a portion of the row in the appropriate bank.

When considering sets of memory chips, the concept of chip selects is added to the equation. Using a chip selects, also known as "PHYSICAL banks", enables the controller to access a certain set of memory modules (up to 1GB for the MSC8156, 2GB for MSC8157 DSPs from Freescale for example) at a time. Once a chip select is enabled, access to the selected memory modules with that chip select are activated, using page selection (rows), banks, and columns. The connection of 2 chip selects is shown in Figure 8 below.

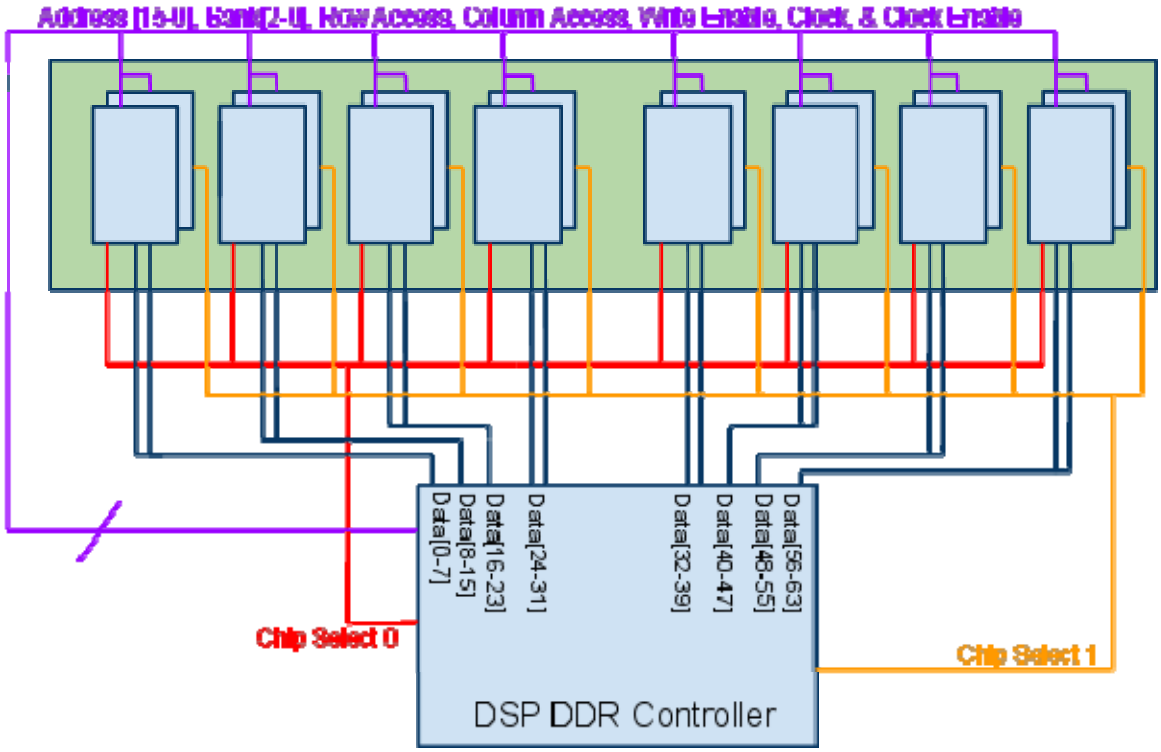


Figure 8: Simplified view: DDR Controller to Memory Connection: 2 Chip Selects

In Figure 8 we see at the bottom we have our DSP device which is intended to access DDR memory. There are a total of 16 chips connected to 2 chip selects: chip select 0 on the left in red, and 1 on the right in orange. The 16 discrete chips are paired such that a pair of chips shares ALL the same signals (Address, bank, data, etc), except for the chip select pin. (Interesting note: This is basically how a dual rank DDR is organized, except each "pair of chips" exists within a single chip). There are 64 data bits. So for a single chip select, when we access DDR and write 64 contiguous bits of data to DDR memory space in our application, the DDR controller doing the following:

1. Selecting chip select based on your address (0 for example)
2. Opening the same page (row) for each bank on all 8 chips using the DDR address bits during the Row Access phase

- New rows are opened via the ACTIVE command, which copies data from the row to a “row buffer” for fast access
- Rows that were already opened do not require an active command and can skip this step

3. During the next phase, the DDR controller will select the same column on all 8 chips. This is the column access phase
4. Finally, the DDR controller will write the 64 bytes to the now open row buffers for each of the 8 separate DDR chips which each input 8 bits.

As there is a command to open rows, there is also one to close rows, called **PRECHARGE**, which tells the DDR modules to store the data from the row buffers back to the actual DDR memory in the chip, thus freeing up the row buffer. So when switching from one row to the next in a single DDR bank, we have to PRECHARGE the open row to close it, and then ACTIVATE the row we wish to start accessing.

A side effect of an ACTIVATE command is that the memory is automatically read and written - thus REFRESHing it. If a row in DDR is PRECHARGED, then it must be periodically refreshed (read/re-written with the same data) to keep data valid. DDR controllers have an autorefresh mechanism that does this for the programmer.

DDR Data Flow Optimization for Power

Now that the basics of DDR accesses have been covered, we can cover how DDR accesses can be optimized for minimal power consumption. As is often the case, optimizing for minimal power consumption is beneficial for performance as well.

The components of DDR power consumption are explained in Micron's Technical Note 41: Calculating Memory System Power for DDR3 [5]. DDR consumes power in all states, even when the CKE (clock enable - enabling the DDR to perform any operations) is disabled, though this is minimal. One technique to minimize DDR power consumption is made available by some DDR controllers which have a power saving mode that de-asserts the CKE pin - greatly reducing power. The Freescale DSP devices, including the MSC8156, call this mode Dynamic Power Management Mode, which can be enabled via the DDR_SDRAM_CFG[DYN_PWR] register. This feature will deassert CKE when no memory refreshes or accesses are scheduled. If the DDR memory has self-refresh capabilities, then this power saving mode can be prolonged as refreshes are not required from the DDR controller.

This power savings mode does impact performance some, as enabling CKE when a new access is scheduled adds a latency delay.

Micron's DDR power calculator can be used to estimate power consumption for DDR. If we choose 1GB x8 DDR chips with -125 speed grade, and we can see estimates for the main power consuming actions on DDR. Power consumption for non-idle operations is additive, so total power is the idle power plus non-idle operations.

- Idle with no rows open and CKE low is shown as: 4.3mW (IDD2p)
- Idle with no rows open and CKE high is shown as: 24.6mW (IDD2n)
- Idle with rows open and no CKE low is shown as: 9.9mW (IDD3p)
- Idle with rows open and CKE high is shown as: 57.3mW (IDD3n)
- ACTIVATE and PRECHARGE is shown as consuming 231.9mW
- REFRESH is shown as 3.9mW
- WRITE is shown as 46.8mW
- READ is shown as 70.9mW

We can see that using the Dynamic Power Management mode saves up to 32mW of power, which is quite substantial in the context of DDR usage.

Also, it is clear that the software engineer must do whatever possible to minimize contributions to power from the main power contributors: ACTIVATE, PRECHARGE, READ, and WRITE operations.

The power consumption from row activation/precharge is expected as DDR needs to consume a considerable amount of power in decoding the actual ACTIVATE instruction and address followed by transferring data from the memory array into the row buffer. Likewise, the PRECHARGE command also consumes a significant amount of power in writing data back to the memory array from row buffers.

Optimizing Power by timing

One can minimize the maximum "average power" consumed by ACTIVATE commands over time by altering the timing between row activate commands, tRC (a setting the programmer can set at start up for the DDR controller). As such, this is an optimization done only once, but seen throughout the time the device is powered. By extending the time required between DDR row activates, the maximum power spike of activates is spread, so the amount of power pulled by the DDR in a given period of time is lessened, though the total power for a certain number of accesses will remain the same. The important thing to note here is that this can help with limiting the maximum (worst case) power seen by the device, which can be helpful when having to work within the confines of a certain hardware limitation (power supply, limited decoupling capacitance to DDR supplies on the board, etc).

Optimizing with Interleaving:

Now that we understand our main enemy in power consumption on DDR is the activate/precharge commands (for both power and performance), we can devise plans to minimize the need for such commands. There are a number of things to look at here, the first being address interleaving, which will reduce ACTIVATE/PRECHARGE command pairs is via interleaving chip selects (physical banks) and additionally by interleaving logical banks.

In setting up the address space for the DDR controller, the row bits and high order mid-order chip select and bank select bits may be swapped to enable DDR interleaving, whereby changing the higher order address enables the DDR controller to stay on the same page while changing chip selects (physical banks) and then changing logical banks before changing rows. The software programmer can enable this in the MSC8156 DSP by enabling the BA_INTLV_CTL bits of the DDR_SDRAM_CFG register. One interleaving by physical and logical bank is enabled, the core-to-DDR bit addressing appears as shown in Figure 9 below.

By interleaving this way, once the 12 bits of column (and LSB) address space are used, logical bank then we will move to the next logical bank to start accessing (without necessarily requiring a PRECHARGE/ACTIVATE). And 15 bits of address space are available using different chip selects if there are multiple chip selects available on the specific board’s memory layout.

Row x Col		MSB		Address from Core Initiator																										LSB
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2–0
15 x 10 x 3	MRAS	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CS SEL														
	MBA																														
	MCAS																														
14 x 10 x 3	MRAS		13	12	11	10	9	8	7	6	5	4	3	2	1	0	CS SEL														
	MBA																	2	1	0											
	MCAS																				9	8	7	6	5	4	3	2	1	0	
14 x 10 x 2	MRAS			13	12	11	10	9	8	7	6	5	4	3	2	1	0	CS SEL													
	MBA																		1	0											
	MCAS																				9	8	7	6	5	4	3	2	1	0	
13 x 10 x 3	MRAS			12	11	10	9	8	7	6	5	4	3	2	1	0	CS SEL														
	MBA																	2	1	0											
	MCAS																				9	8	7	6	5	4	3	2	1	0	
13 x 10 x 2	MRAS				12	11	10	9	8	7	6	5	4	3	2	1	0	CS SEL													
	MBA																		1	0											
	MCAS																				9	8	7	6	5	4	3	2	1	0	

Figure 9: 64-bit DDR memory with chip select and logical bank interleaving [6]

Optimizing Memory Software Data Organization

We also need to consider the layout of our memory structures within DDR. In the case of using large ping-pong buffers for example, the buffers may be organized so that each buffer is in its own logical bank. This way, if DDR is not interleaved, we still can avoid unnecessary ACTIVATE/PRECHARGE pairs in the case that a pair of buffers is larger than a single row (page).

Optimizing General DDR Configuration

There are other features available to the programmer, which can positively or negatively affect power, including “open/closed” page mode. Closed page mode is a feature available in some controllers which will perform an auto-precharge on a row after each read or write access. This of course unnecessarily increases the power consumption in DDR as a programmer may need to access the same row 10 times for example, closed page mode would yield at least 9 unneeded PRECHARGE / ACTIVATE command pairs.

In the example DDR layout discussed above, this could consume an extra $231.9mW * 9 = 2087.1mW$.

As you may expect, this has an equally negative effect on performance due to the stall incurred during memory PRECHARGE and ACTIVATE.

Optimizing DDR Burst Accesses

DDR technology has become more restrictive with each generation, specifically in regards to how data bursts are organized. Bursts, as they are described in hardware, is a chunk of data sent consecutively clock cycle after clock cycle (as opposed to having clock cycles where no data is sent). Since DDR can send data on both edges of a clock cycle, there is an additional term here, called a “beat”, which refers to a single edge of the clock. Using this we can describe how long our bursts are in “half clock cycles” or “beats. In regards to how DDR has become more restrictive: DDR2 allows 4 beat burst and 8 beat bursts, whereas DDR3 only allows 8. This means that DDR3 will treat all burst lengths as 8 beat (bursts of 8 accesses long). So for the 8 byte (64 bit) wide DDR accesses we have been discussing here, single burst accesses are expected to be 8 beats of 8 bytes, or 64 bytes long.

If accesses are not 64 bytes wide, there will be stalls due to the hardware design. This means that if the DDR memory is accessed for only reading (or writing 32 bytes of data at a time, DDR will only be running at 50% efficiency, as the hardware will still perform reads/writes for the full 8 beat burst, though only 32 bytes will be used. Because DDR3 operates this way, the same amount of power is consumed whether doing 32 byte or 64byte long bursts to our memory here. So for the same amount of data, if doing 4 beat (32 byte) bursts, the DDR3 would consume approximately twice the power.

The recommendation here then is to fill all accesses to DDR to be full 8 beat bursts in order to maximize power efficiency. To do this, the programmer must be sure to **pack data in the DDR so that accesses to the DDR are in at least 64 byte wide chunks**. Packing data to be 64 byte aligned or any other alignment can be done through the use of pragmas, for example, in Freescale processors. These pragmas, when declared with a variable, force the compiler to align the variable to a certain width.

The concept of data packing can be used to reduce the amount of used memory as well. For example, packing 8 single bit variables into a single character reduces memory footprint and increases the amount of usable data the core or cache can read in with a single burst.

In addition to data packing, **accesses need to be 8 byte aligned** (or aligned to the burst length). If an access is not aligned to the burst length, for example, if on the MSC8156, a 8 byte access starts with a 4 byte offset, both the first and second access will effectively become 4-beat bursts, reducing bandwidth utilization to 50% (instead of aligning to the 64 byte boundary and reading data in with 1 single burst).

SRAM AND CACHE DATA FLOW OPTIMIZATION FOR POWER

Another optimization related to the usage of off chip DDR is avoidance: avoiding using external off chip memory, and maximizing accesses to internal on-chip memory saves the additive power draw that occurs when activating not only internal device buses and clocks, but also off chip buses, memories arrays, etc.

High speed memory close to the DSP processor core is typically SRAM memory, whether it functions in the form of cache or as a local on-chip memory. SRAM differs

from SDRAM in a number of ways (such as no ACTIVATE/PRECHARGE, and no concept of REFRESH), but some of the principles of saving power still apply, such as pipelining accesses to memory via data packing and memory alignment.

The general rule for SRAM access optimization is that accesses should be optimized for higher performance. The fewer clock cycles the device spends doing a memory operation = less time that memory, buses, and core are all activated for said memory operation.

SRAM (All Memory) and Code Size

As a programmer, we can affect this in both program and data organization. Programs may be optimized for minimal code size (by a compiler, or by hand), in order to consume a minimal amount of space. Smaller programs require less memory to be activated to read the program. This applies not only to SRAM, but also DDR and any type of memory - the less memory that has to be accessed implies the less amount of power drawn.

Aside from optimizing code using the compiler tools, other techniques such as instruction packing, which are available in architectures like the SC3850, enable fitting maximum code into a minimum set of space. The VLES (Variable Length Execution Set) instruction architecture allows the program to pack multiple instructions of varying sizes into a single execution set. As execution sets are not required to be 128 bit aligned, instructions can be packed tightly, and the SC3850 prefetch, fetch, and instruction dispatch hardware will handle reading the instructions and identifying start and end of each instruction set (via instruction prefix encodings prepended in machine code by the StarCore assembler tools).

Additionally, size can be saved in code by creating functions for common tasks. If tasks are similar, consider use the same function with parameters passed that determine the variation to run instead of duplicating the code in software multiple times.

Be sure to make use of combined functions where available in the hardware. In the Freescale StarCore architecture, using a Multiply Accumulate (MAC) instruction, which takes 1 pipelined cycle, saves space and performance in addition to power over using separate multiple and add instructions.

Some hardware provides code compression at compile time and decompression on the fly, so this may be an option depending on the hardware the user is dealing with. The problem with this strategy is related to the size of compression blocks. If data is compressed into small blocks, then not as much compression optimization is possible, but this is still desirable over the alternative. During decompression, if code contains many branches or jumps, the processor will end up wasting bandwidth, cycles, and power decompressing larger blocks that are hardly used.

The problem with the general strategy of minimizing code size is the inherent conflict between optimizing for performance and space. Optimizing for performance generally does not always yield the smallest program, so determining ideal code size vs cycle performance in order to minimize power consumption requires some balancing and profiling. The general advice here is to use what tricks are available to minimize code size without hurting the performance of a program that meets real time requirements. The 80/20 rule of applying performance optimization to the 20% of code that performs 80% of the work, while optimizing the remaining 80% of code for size is a good practice to follow.

SRAM Power Consumption and Parallelization

It is also advisable to optimize data accesses in order to reduce the cycles in which SRAM is activated, pipelining accesses to memory, and organizing data so that it may be accessed consecutively. In systems like the MSC8156, the core / L1 caches connect to the M2 memory via a 128-bit wide bus. If data is organized properly, this means that 128 bit data accesses from M2 SRAM could be performed in one clock cycle each, which would obviously be beneficial when compared to doing 16 independent 8 bit accesses to M2 in terms of performance and power consumption.

An example showing how one may use move instructions to write 128 bits of data back to memory in a single instruction set (VLES) is provided below:

```
[
MOVERH.4F d0:d1:d2:d3,(r4)+n0
MOVERL.4F d4:d5:d6:d7,(r5)+n0
]
```

We can parallelize memory accesses in a single instruction (as with the above where both of the moves are performed in parallel), and even if the accesses are to separate memories or memory banks, the single cycle access still consumes less than the power of doing two independent instructions in two cycles.

Another note: as with DDR, SRAM accesses need to be aligned to the bus width in order to make full use of the bus.

Data Transitions and Power Consumption

SRAM power consumption may also be affected by the TYPE of data used in an application. Power consumption is affected by the number of data transitions (from 0's to 1's) in memory as well. This power effect also trickles down to the DSP core processing elements as well, as found by Kojima, et al [7]. Processing mathematical instructions using constants consumes less power at the core than with dynamic variables. In many

devices, because pre-charging memory to reference voltage is common practice in SRAM memories, power consumption is also proportional to the number of zero's as the memory is pre-charged to a high state [7].

Using this knowledge, it goes without saying that re-use of constants where possible, and avoiding zero-ing out memory unnecessarily will, in general, save the programmer some power.

CACHE UTILIZATION AND SOC MEMORY LAYOUT

Cache usage can be thought of in the opposite manner to DDR usage when designing a program. An interesting detail about cache is: both dynamic and static power increase with increasing cache sizes, however, the increase in dynamic power is small. The increase in static power is significant, and becomes increasingly relevant for smaller feature sizes, as noted in [8]. As the software programmer, we have no impact on the actual cache size available on a device, but when it is provided, based on the above, it is our duty to use as much of it as possible!!!

For SoC level memory configuration and layout, optimizing the most heavily used routines and placing them in the closest cache to the core processors will offer not only the best performance, but also better power consumption.

Explanation of Locality

The reason the above is true is thanks to the way caches work. There are a number of different cache architectures, but they all take advantage of the principle of locality. The principle of locality basically states that if one memory address is accessed, the probability of an address nearby being accessed soon is relatively high. Based on this, when a cache miss occurs (when the core tries to access memory that has not been brought into the cache), the cache will read the requested data in from higher level

memory one line at a time. This means that if the core tries to read a 1 byte character from cache, and the data is not in the cache, then there is a miss at this address. When the cache goes to higher level memory (whether it be on-chip memory or external DDR, etc), it will not read in an 8 bit character, but rather a full cache line. If our cache uses cache sizes of 256 bytes, then a miss will read in our 1 byte character, along with 255 more bytes that happen to be on the same line in memory.

This is very effective in reducing power if used in the right way. If we are reading an array of characters aligned to the cache line size, once we get a miss on the first element, although we pay a penalty in power and performance for cache to read in the first line of data, the remaining 255 bytes of this array will be in cache. When handling image or video samples, a single frame would typically be stored this way, in a large array of data. When performing compression or decompression on the frame, the entire frame will be accessed in a short period of time, thus it is spatially and temporally local.

In the case of the MSC8156, there are two levels of cache for each of the 6 DSP processor cores: L1 cache (which consists of 32KB of instruction and 32KB of data cache), and a 512KB L2 memory which can be configured as L2 cache, or M2 memory. At the SoC level, there is a 1MB memory shared by all cores called M3. L1 cache runs at the core processor speed (1GHz), L2 cache effectively manages data at the same speed (double the bus width, half the frequency), and M3 runs at up to 400MHz. The easiest way to make use of the memory hierarchy is to enable L2 as cache and make use of data locality. As discussed above, this works when data stored with high locality. Another option is to DMA data into L2 memory (configured in non-cache mode). We will discuss DMA in a later section.

When we have a large chunk of data stored in M3 or in DDR, the MSC8156 can draw this data in through the caches simultaneously. L1 and L2 caches are linked, so a miss from L1 will pull 256Bytes of data in from L2, and a miss from L2 will pull data in at 64Bytes at a time (64B line size) from the requested higher level memory (M3 or DDR). Using L2 cache has two advantages over doing directly to M3 or DDR. First, it is running at effectively the same speed as L1 (though there is a slight stall latency here, it is negligible), and second, in addition to being local and fast, it can be up to 16 times larger than L1 cache, allowing us to keep much more data in local memory than just L1 alone would.

Explanation of Set-Associativity

All caches in the MSC8156 are 8 way set-associative. This means that the caches are split into 8 different sections ("ways"). Each section is used to access higher level memory, meaning that a single address in M3 could be stored in one of 8 different sections (ways) of L2 cache for example. The easiest way to think of this is that the section (way) of cache can be overlaid onto the higher level memory x times. So so if L2 is set up as all cache, the following equation calculates how many times each set of L2 memory is overlaid onto M3:

$$\#\ of\ overlays\ O = \frac{M3size}{(L2size\ /\ 8\ ways)}$$

$$= \frac{1MB}{(512KB\ /\ 8)} = 16384\ overlays$$

In the MSC8156, a single way of L2 cache is 64KB in size, so addresses are from 0x0000_0000 to 0x0001_0000 hexidecimal. If we consider each way of cache

individually, we can explain how a single way of L2 is mapped to M3 memory. M3 addresses start at 0xC000_0000. So M3 addresses 0xC000_0000, 0xC001_0000, 0xC002_0000, 0xC003_0000, 0xC004_0000, etc (up to 16K times) all map to the same line of a way of cache. So if way #1 of L2 cache has valid data for M3's 0xC000_0000, and the core processor wants to next access 0xC001_0000, what is going to happen?

If the cache has only 1 way set associativity, then the line of cache containing 0xC000_0000 will have to be flushed back to cache and re-used in order to cache 0xC001_0000. In an 8 way set associative cache, however, we can take advantage of the other 7 x 64KB sections "ways" of cache. So we can potentially have 0xC000_0000 stored in way #1, and the other 7 ways of cache have their first line of cache as empty. In this case, we can store our new memory access to 0xC001_0000 in way #2.

So, what happens when there is an access to 0xC000_0040? (0x40 == 64B). The answer here is that we have to look at the 2nd cache line in each way of L2 to see if it is empty, as we were only considering the 1st line of cache in our example above. so here we now have 8 more potential places to store a line of data (or program).

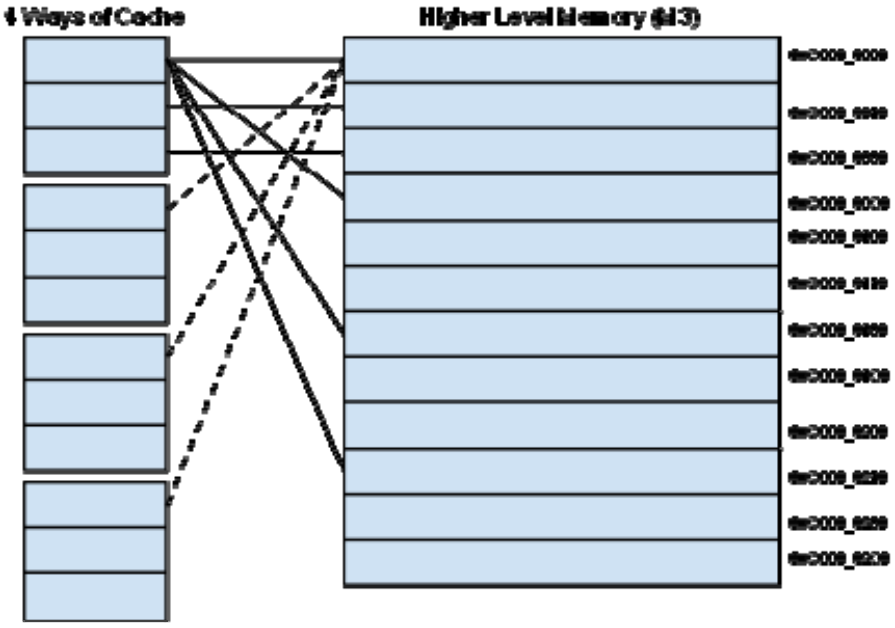


Figure 10: Set Associativity by Cache Line: 4 way set associative cache

Figure 10 above shows a 4 way set associative cache connecting to M3. In this figure, we can see that every line of M3 maps to 4 possible lines of the cache (one for each way). So line 0xC000_0040 maps to the 2nd line (second "set") of each way in the cache. So when the core wants to read 0xC000_0040, but the first way has 0xC000_0100 in it, the cache can load the cores request into any of the other three ways if their 2nd line is empty (invalid).

The reason for discussing set associativity of caches is that it does have some effect on power consumption (as one might imagine). The goal for maximizing power consumption (and performance) when using cache is to maximize the hit rate in order to minimize accesses to external buses and hardware caused by misses. Set-associativity is normally already determined by hardware, but in the case that the programmer can change set associativity: set-associative caches maintain a higher hit-rate than directly mapped caches, and thus draw lower power.

Memory Layout for Cache

While having an 8-way set associative architecture is statistically beneficial in improving hit ratio and power consumption, the software programmer may also directly improve hit ratio in the cache, and thus lower power by avoiding conflicts in cache. Conflicts in cache occur when the core needs data that will replace cache lines with currently valid data that will be needed again.

We can organize memory in order to avoid these conflicts in a few different ways. For memory segments we need simultaneously, it is important to pay attention to the size of ways in the cache. In our 8 way L2 cache, each way is 64KB. As we discussed before, we can simultaneously load 8 cache lines with the same lower 16 bits of address (0x0000_xxxx).

Another example is if we are working with 9 arrays with 64KB of data simultaneously. If we organize each array contiguously data will be constantly thrashed as all arrays share the same 64KB offset. If the same indices of each array are being accessed simultaneously, we can offset the start of some of the arrays by inserting buffer, so that each array does not map to the same offset (set) within a cache way.

When data sizes are larger than a single way, the next step is to consider reducing the amount of data that is pulled into the cache at a time - process smaller chunks at a time.

Write Back vs Write Through Caches

Some caches are designed as either “write back” or “write through” caches, and others, such as the MSC815x series DSPs are configurable as either. Write back and write

through buffering differs in how data from the core is managed by the cache in the case of writes.

Write-back is a cache writing scheme in which data is written only to the cache. The main memory is updated when the data in the cache is replaced. In the write-through cache write scheme, data is written simultaneously to the cache and to memory. When setting up cache in software, we have to weigh the benefits of each of these. In a multicore system, coherency is of some concern, but so is performance, and power. Coherency refers to how up-to-date data in main memory is compared to the caches. The greatest level of multicore coherency between internal core caches and system level memory is attained by using write-through caching, as every write to cache will immediately be written back to system memory keeping it up to date. There are a number of down sides to write-through caching including:

- Core stalls during writes to higher level memory
- Increased bus traffic on the system buses (higher chance for contention and system level stalls)
- Increased power consumption as the higher level memories and buses are activated for every single memory write

The write-back cache scheme on the other hand, will avoid all of the above disadvantages at the cost of system level coherency. For optimal power consumption, a common approach is to use the cache in write-back mode, and strategically flush cache lines/segments when the system needs to be updated with new data.

Cache Coherency Functions

In addition to write-back and write-through schemes, specific cache commands should also be considered. Commands include:

- Invalidation sweep: invalidating a line of data by clearing valid and dirty bits (effectively just re-labeling a line of cache as “empty”).
- Synchronization sweep: writing any new data back to cache and removing the dirty label.
- Flush sweep: writing any new data back to cache and invalidating the line
- Fetch: fetch data into the cache

Generally these operations can be performed either by cache line, a segment of the cache, or as a global operation. When it is possible to predict that a large chunk of data will be needed in the cache in the near future, performing cache sweep functions on larger segments will make better use of the full bus bandwidths and lead to fewer stalls by the core. As memory accesses all require some initial memory access setup time, but after setup, bursts will flow at full bandwidth, making use of large prefetches will save power when compared to reading in the same amount of data line by line so long as this is done strategically so as to avoid the data we want from being thrashed before the core actually gets to use it.

When using any of these instructions, we have to be careful about the affect it has on the rest of the cache. For instance, performing a fetch from higher level memory into cache may require replacing contents currently in the cache. This could result in thrashing data in the cache and invalidating cache in order to make space for the data being fetched.

In many DSPs, it is possible to lock code or data into the cache or low level memories so that it will always be available when needed. Other options to consider here include code overlay, where segments of code are heavily used only for specific periods of time. This could be the case for a media player, where an MPEG4 decoder segment would be needed for 1 movie, but H.264 may be needed for the next. In this case we could either use cache locking for cache, or DMA for high speed local SRAM in order to bring in relevant functions for the periods of time they are needed in 1 shot. In this case there is some overhead for the large one shot transfer, but in the application example listed – the media player, the setup time for the DMA or instruction cache flush and lock in would be acceptable.

Compiler Cache Optimizations

In order to assist with the above, compilers may be used to optimize cache power consumption by re-organizing memory or memory accesses for us. Two main techniques available are array merging and loop interchanging, explained below [1].

Array merging

Array merging organizes memory so that arrays accessed simultaneously will be at different offsets (different "sets") from the start of a way. Consider the following two array declarations below:

```
int array1[ array_size ];
int array2[ array_size ];
```

The compiler can merge these two arrays as shown below:

```
struct merged_arrays
{
     int array1;
     int array2;
} new_array[ array_ size ]
```

Loop interchanging

In order to re-order the way that high level memory is read into cache: reading in smaller chunks to reduce the chance of thrashing loop interchanging can be used. Consider the code below:

```
for (i=0; i<100; i=i+1)
 for (j=0; j<200; j=j+1)
      for (k=0; k<10000; k=k+1)
          z[ k ][ j ] = 10 * z[ k ][ j ];
```

By **interchanging** the second and third nested loops, the compiler can produce the following code, decreasing the likelihood of unnecessary thrashing during the innermost loop.

```
for (i=0; i<100; i=i+1)
      for (k=0; k<10000; k=k+1)
           for (j=0; j<200; j=j+1)
               z[ k ][ j ] = 10 * z[ k ][ j ];
```

Loop tiling

In some cases, loop interchanging may not be possible, but like interchanging, loop tiling may also be used by the compiler in order to reduce the amount of data being pulled into the cache at one time in order to avoid unnecessary thrashing. Loop tiling effectively breaks loops into smaller segments so the data needed by the innermost loop may in (a portion of) the cache. For example, consider the following simple loop used to assign an array of 32bit int's which would require 80KB:

```
for (k=0; k<10000; k=k+1)
     x[ k ] = 10 * y[ k ];
```

We can reduce the inner loop's footprint to a given block size. In the below case, we set the block size to 200:

```
for (k=0; k<10000; k=k+200)
```

```
for (j=k; j<k+200; j=j+1)
     x[ j ] = 10 * y[ j ];
```

So in this case – we only need 200 array elements (or whatever our block size may be) per variable used in the inner loop available in cache at a time.

PERIPHERAL/COMMUNICATION UTILIZATION

When considering reading and writing of data, of course, we cannot just think about memory access: we need to pull data in and off from the device as well. As such, for the final portion of data path optimization we will look at how to minimize power consumption in commonly used DSP (I/O) peripherals.

Things to consider include the peripheral's burst size, speed grade, transfer width, and general communication modes. Main standard forms of peripheral communication for DSPs include DMA (direct memory access), SRIO (Serial Rapid I/O), Ethernet, PCI Express, and RF antenna interfaces. I2C and UART are also commonly used, though mostly for initialization and debug purposes.

The fact that communication interfaces usually require their own PLLs / clocks increases the individual power consumption impact. The higher clocked peripherals that we need to consider as the main power consumers are the DMA, SRIO, Ethernet, and PCI Express. Clock gating and peripheral low power modes for these peripherals were already discussed in the Low Power Modes section of this paper, so this section will talk about how to optimize actual usage.

Although each protocol is different for the I/O peripherals and the internal DMA, they all share the fact that they are used to read/write data. As such, one basic goal is to maximize the throughput while the peripheral is active in order to maximize efficiency

and the time the peripheral / device can be in a low power state, thus minimizing the active clock times.

The most basic way to do this is to increase transfer / burst size. For DMA, the programmer has control over burst size and transfer size in addition to start / end address (and can follow alignment and memory accessing rules we discussed earlier subsections of Data Path Optimization). Using the DMA, the programmer can not only decide the alignment, but also the transfer "shape" for lack of a better word. What this means is that using the DMA, the programmer can transfer blocks in the form of 2-dimensional, 3-dimensional, and 4-dimensional data chunks, thus transferring data types specific to specific applications on the alignment chosen by the programmer without spending cycles transferring unnecessary data. Figure 11 below demonstrates the data structure of a 3-dimensional DMA.

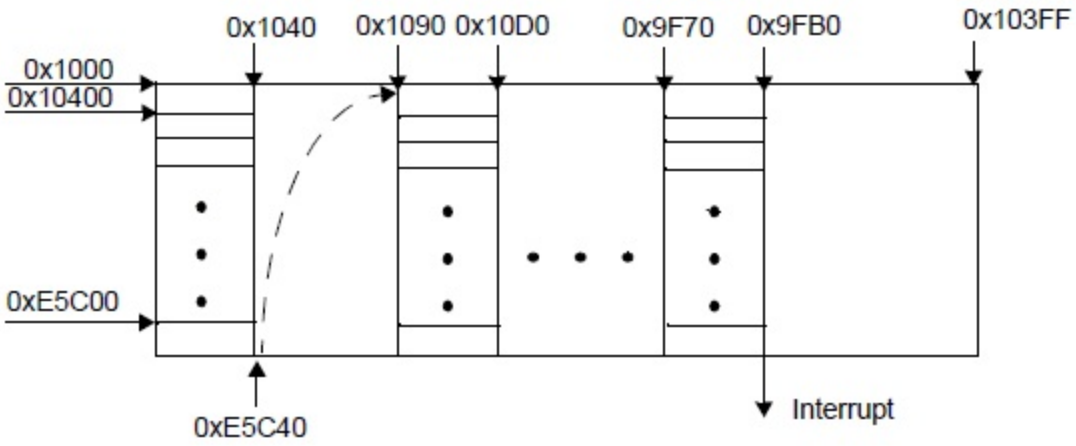


Figure 11: Three Dimensional DMA Data Format

The user programs the start address for data, the length for the first dimension, the offset for the second dimension, the number of transfers, followed by the offset for the 3rd dimension and the number of transfers. At the end of the all transfers, the programmer may also program the DMA to interrupt the core to signal data transfer

completion. Having the DMA intelligently moving the data in the format and organization needed by the user's application helps optimize data flow and core processing by avoiding the need for the core to re-organize data or alter algorithms that are optimized for data in a specific format. This also simplifies the maintaining of certain alignments as the programmer can decide where each dimension of the data structure starts.

Other high speed peripherals generally will also use a DMA, whether it be the system DMA, or the peripheral's own private DMA for data passing. In the case of the MSC8156, the SRIO, PCI Express, and Ethernet controllers all have their own DMAs separate from the system DMA for data transfers. The basics still apply here: we want data transfers to be long (long bursts), we want bus accesses to be aligned, and additionally, one more thing we want is optimal access to the system bus! We will discuss system bus optimization later in this section.

DMA of Data vs CPU

While on the topic of DMA, we need to consider whether the core should move data from internal core memory or if a DMA should be utilized in order to save power. As the DMA hardware is optimized solely for the purpose of moving data, it will move data while consuming less power than the core (which is generally running at much higher frequencies than the DMA). As the core runs at such a higher frequency, is not intended solely for data movement, etc, the core utilizes more dynamic power while incurring heavy stall penalties when accessing external memory.

As some external memory access and stalls are incurred for writing to peripheral registers when setting up the DMA, there is a point where data accesses are too small or infrequent to justify DMA usage. In general, when moving larger chunks of data, or data

in a predictable manner, for the purpose of power consumption (and core performance), DMA should be utilized for maximum power savings and application efficiency.

For transactions and I/O that are not large enough to justify DMA, we can consider caching as this assists with stalls and requires virtually no core intervention. Generally speaking, using the cache is much simpler than DMA, so this is the generally accepted solution for unpredictable data I/O, while DMA should be used for larger memory transfers. Due to the overhead for programming the DMA, and the unique properties of data per application, the trade-off between power savings, performance, and program complexity from DMA to cache has to be done on a case by case basis. Peripherals with their own DMA generally require the programmer to use that DMA for peripheral interaction, which is a good habit to force the programmer into, as we have just discussed.

Coprocessors

Just as the DMA peripheral is optimized for data movement and can do so more efficiently with less power consumption than the high frequency DSP core, so are other peripherals acting as coprocessors able to perform special functions more efficiently than the DSP core. In the case of the MSC8156, the MAPLE baseband coprocessor includes hardware for Fast Fourier Transforms, Discrete Fourier Transforms, and Turbo Viterbi. When a chain of transforms can be offloaded onto the MAPLE, depending on the cost of transferring data and transform sizes, the system is able to save power and cycles by offloading the core and having the MAPLE coprocessor do this work as the MAPLE is running at a much lower frequency than the core, has fewer processing elements aimed at a single function, and also has automatic lower power modes that are used when the MAPLE is not processing transforms, etc.

System Bus Configuration

System bus stalls due to lack of priority on the bus can cause a peripheral to actively wait unnecessarily for extra cycles when not set up properly. These extra active wait cycles mean more wasted power. Generally DSPs will have system bus configuration registers which allow the programmer to configure the priority and arbitration per bus initiator port. In the case of the MSC8156, the system bus (called the CLASS) has 11 initiator ports and 8 target ports (memory and register space). When the programmer understands the I/O needs of the application, it is possible to set up priority appropriately for the initiators that need the extra bandwidth on the bus so they can access memory and register space with minimal stalls.

There is not much of a trick to this, simply set priority based on I/O usage. Some devices such as the MSC815x series DSPs provide bus profiling tools which enable the programmer to count the number of accesses per initiator port to each target. This allows the programmer to see where congestion and bottlenecks are occurring in order to appropriately configure and tune the bus. Profiling tools also allow the programmer to see how many “priority upgrades” are needed per port. This means that the programmer can temporarily assign test priorities to each port, and if some ports are constantly requiring priority upgrades, the programmer can decide to set the starting priority of these ports one level up and re-profile.

Peripheral Speed Grades and Bus Width

Like with the system bus access, the peripheral’s external interface should be set up according to the actual needs of the system. The catch-22 of I/O peripherals is that some peripherals require being powered on all the time (so minimal to no use of low power modes is available). If a communication port such as SRIO is dedicated to receiving incoming blocks of data to process, when no data is coming in, clocks and low

power modes for the SRIO port are not an option. As such, there is a balancing game to be played here.

In testing software and power consumption, we found on the MSC8156 that running 4 lanes of SRIO at 3.125GHz with 40% utilization (~4Gbps of data) consumes a comparable amount, or even less power as running 4 lanes of SRIO at 2.5GHz with 50% utilization (the same data throughput). So the user needs to test various cases or make use of the device manufacturer's power calculator in order to make an informed decision. In a case like this, peripherals which have an auto-idle feature should make use of the higher speed bus in order to maximize sleep times.

SRIO, PCI Express, Ethernet over SGMII, and some antenna interfaces make use of the same serial I/O hardware, so similar care should be taken here. All could be required to be held in an active mode as a form of "device wake" or signaling to the DSP core, meaning they may be restricted from going into sleep modes. In the case of antenna signaling, this is especially detrimental as the active antenna's RF interface has to constantly consume power to emit signal. If possible, it is ideal to use an alternative method for waking the DSP core in order to enable idle and sleep modes on the antenna.

Peripheral to Core Communication

When considering device waking, and general peripheral to core I/O, we have to consider how the peripheral interacts with the core processors. How does the core know that data is available? How often is the core notified of data available? How does the core know when to send data over the peripheral? There are three main methods for managing this: polling, time based processing, and interrupt processing

Polling

Polling is by far the least efficient method of core-peripheral interaction as it has the core constantly awake and burning through high frequency clock cycles (consuming active current) just to see if data is ready. The only positive of using this method happens when the programmer is not concerned about power consumption. In this case, polling enables the core to avoid context switches that occur during interrupt processing, thus saving some cycles in order to access data faster. Generally this is only used for testing maximum peripheral bandwidth as opposed to being used in a real application.

Time based processing

Time based processing works on the assumption that data will always be available at a certain interval. For example, if a DSP is processing a GSM voice codec (AMR, EFR, HR, etc). The core will know that samples will be arriving every 20ms, so the core can go look for the new audio samples on this time basis and not poll. This process allows the core to sleep, and use a timer for wake functionality, followed by performing data processing. The down side of this is the complexity and inflexibility of this model: setup and synchronization requires a lot of effort on the programmer's side, and the same effect can be reached using simple interrupt processing.

Interrupt processing

The final core to peripheral communication mechanism is also the most commonly used one as it allows the benefits of the time based processing without the complicated software architecture. We also briefly discussed using interrupt processing in the Low Power Modes section as a method for waking the core from sleep states: when new data samples and packets come in for processing, the core is interrupted by the peripheral (and can be woken if in sleep state) to start processing new data. The

peripheral can also be used to interrupt the core when the peripheral is ready to transfer new data, so that the core does not have to constantly poll a heavily loaded peripheral to see when data is ready to send.

Power consumption results for polling vs interrupt processing are already shown in Figure 6 when comparing the Baseline MJPEG vs the Using WAIT for PD and Using STOP for PD modes. When not using WAIT and STOP modes, the application would constantly check for new buffers without taking advantage of massive idle times in the application.

Section 3.3: Algorithmic

Of the three main areas of power optimization discussed here, algorithmic optimization requires the most work for a given amount of power savings. Algorithmic optimization includes: optimization at the core application level, code structuring, data structuring (in some cases, this could be considered as data path optimization), data manipulation, and optimizing instruction selection.

Compiler Optimization Levels

In the data path section, we briefly discussed that the compiler can be used to optimize code for minimal size. The compiler may also be used to optimize code for maximum performance (utilizing the maximum number of processing units per cycle and minimizing the amount of time code is running). This is also discussed in [9], where the TI C6000 DSP is used to test whether optimizing for performance will reduce power consumption. As expected, the results show that increasing the number of processing units will increase the power consumed per cycle, but the total power to perform a function over time will reduce as the number of cycles to perform the function is reduced. The question of when to optimize for performance versus code size generally still fits with the 80/20 rule (80% of cycle time is spent in 20% of the code), so as mentioned in the data path section, the general rule here is to optimize the cycle hungry (20%) portion of code for performance, while focusing on minimizing code size for the rest. Fine tuning this is the job of the programmer and will require power measurement (as discussed in section 2). The rest of this section will cover specific algorithmic optimizations, some of which may be performed by the performance optimizer in the compiler.

Instruction Packing

Instruction packing was already listed in the data path optimization section above, but may also be listed as an algorithmic optimization as it involves not only how memory is accessed, but also how code is organized. Refer to SRAM and Cache Data Flow Optimization for Power in this paper for details on instruction packing.

Loop Unrolling

We briefly discussed using altering loops in code in order to optimize cache utilization before. Another method for optimizing both performance and power in DSP processors is via loop-unrolling, discussed under DSP Software Code Optimization. This method effectively partially unravels a loop, as shown in the code snippets below:

Regular loop:

```
for (i=0; i<100; i=i+1)
      for (k=0; k<10000; k=k+1)
          a[ i ]= 10 * b[ k ];
```

Loop unrolled by 4x:

```
for (i=0; i<100; i=i+4)
      for (k=0; k<10000; k=k+4)
          {
               a[ i ]= 10 * b[ k ];
               a[ i+1 ]= 10 * b[ k+1 ];
               a[ i+2 ]= 10 * b[ k+2 ];
               a[ i+3 ]= 10 * b[ k+3 ];
          }
```

Unrolling code in this manner enables the compiler to make use of 4 MACs (Multiply-Accumulates) in each loop iteration instead of just one, thus increasing processing parallelization and code efficiency (more processing per cycle means more idle cycles available for sleep and low power modes). In the above case, we increase the parallelization of the loop by four times, so we perform the same amount of MACs in ¼

the cycle time, thus the effective active clock time needed for this code is reduced by 4x. Measuring the power savings using the MSC8156, power consumption, we find that the above example optimization (saving 25% cycle time by utilizing 4 MACs per cycle instead of one enables the core a ~48% total power savings over the time this routine is executed).

Completely unrolling loops is not advisable as it is counterproductive to code size minimization efforts we discussed in the data path section, which would lead to extra memory accesses and possibility of increased cache miss penalties.

Software Pipelining

Another technique common to both DSP performance optimization and DSP power optimization is software pipelining. Software pipelining is a technique where the programmer splits up a set of interdependent instructions that would normally have to be performed one at a time so that the DSP core can begin processing multiple instructions in each cycle. Rather than explaining in words, the easiest way to follow this technique is to see an example.

Say we have the following code segment:

Regular Loop:

```
for (i=0; i<100; i=i+1)
{
     a[ i ]= 10 * b[ i ];
     b[ i ]= 10 * c[ i ];
     c[ i ]= 10 * d[ i ];
}
```

Right now, although we have 3 instructions occurring per loop, the compiler will see that the first instruction depends on the second instruction, and thus could not be

pipelined with the second, nor can the second instruction be pipelined with the third due to interdependence: a[i] cannot be set to b[i] as b[i] is simultaneously being set to c [i], and so on. So right now the DSP processor has to execute the above loop 100 times with each iteration performing 3 individual instructions per cycle (not very efficient), for a total of 300 cycles (best case) performed by MACs in the core of the loop. With software pipelining, we can optimize this in the following way:

First we see where we can parallelize the above code by unrolling the loop some:

Unrolled loop:

```
a[ i ]= 10 * b[ i ];
b[ i ]= 10 * c[ i ];
c[ i ]= 10 * d[ i ];
     a[i+1]= 10 * b[i+1];
     b[i+1]= 10 * c[i+1];
     c[i+1]= 10 * d[i+1];
          a[i+2]= 10 * b[i+2];
          b[i+2]= 10 * c[i+2];
          c[i+2]= 10 * d[i+2];
               a[i+3]= 10 * b[i+3];
               b[i+3]= 10 * c[i+3];
               c[i+3]= 10 * d[i+3];
```

Using the above, we can see that certain instructions that are not interdependent. The first assignment of array “a” relies on the original array “b”, meaning we can potentially assign a entirely before doing any other instructions. If we do this, this means that array “b” would be entirely free of dependence and could be completely assigned to the original array “c”. We can abstract this for c as well.

We can use this idea to break the code apart and add parallelism by placing instructions together that can run in parallel when doing some assignment in advance:

First, we have to perform our first instruction (no parallelism):

```
a[ i ]= 10 * b[ i ];
```

Then we can have two instructions performed in one cycle:

```
b[ i ]= 10 * c[ i ];
a[i+1]= 10 * b[i+1];
```

Here we see that the first and second lines do not depend on each other, so there is no problem with running the above in parallel as one execution set.

Finally, we reach the point where three instructions in our loop are all being performed in one cycle:

```
c[ i ]= 10 * d[ i ];
b[i+1]= 10 * c[i+1];
a[i+2]= 10 * b[i+2];
```

Now we see how to parallelize the loop and pipeline, the final software pipelined will first have some “setup”, also known as loading the pipeline. This consists of the first sets of instructions we performed above. After this we have our pipelined loop:

```
//pipeline loading – first stage
a[ i ]= 10 * b[ i ];
//pipeline loading – second stage
b[ i ]= 10 * c[ i ];
a[i+1]= 10 * b[i+1];
//pipelined loop
for (i=0; i<100-2; i=i+1)
{
     c[ i ]= 10 * d[ i ];
     b[i+1]= 10 * c[i+1];
     a[i+2]= 10 * b[i+2];
}
//after this, we still have 2 more partial loops:
c[i+1]= 10 * d[i+1];
b[i+2]= 10 * c[i+2];
//final partial iteration
c[i+2]= 10 * d[i+2];
```

By pipelining the loop, we enabled the compiler to reduce the number of cycles for MACs from 300 to:

- 3 MACs that can be performed in 2 cycles for pipeline loading
- 100 cycles (3 MACs each) in the core of our loop
- 3 MACs that can be performed in 2 cycles for pipeline loading

For a total of 104 cycles or roughly 1/3 of the execution time, thus reducing the amount of time the core clocks must be active by 3x for the same functionality! Similar to the loop unrolling case, the pipelining case has enabled us to save substantially: ~43% total power over the time this routine is executed.

Eliminating Recursion

An interesting technique suggested by [10] is that we want to eliminate recursive procedure calls in order to reduce function call overhead.

Recursive procedure calls require the functions general context, etc to be pushed onto the stack with each call. So in the classic case of the factorial example (n!), this can be calculated using recursion with a function as follows:

$fn!(0) = 1$ For n==0
$fn!(n) = fn!(n-1);$ For n > 0

If this recursive factorial function is called with n=100, there would be ~100 function calls entailing 100 branch to subroutines (which are change of flow routines which affect the program counter and software stack). Each change of flow instruction takes longer to execute because not only is the core pipeline disrupted during execution, but every branch adds at least a return address to the call stack. Additionally, in the case that multiple variables are being passed, this also must be pushed onto the stack.

This means that this recursive subroutine requires 100x individual writes to physical memory and related stall as writes/reads to memory will not be pipelined and 100x pipeline stalls due to change of flow.

We can optimize this by moving to a simple loop

```
int res=1;
for(int i=0; i<n; i++)
{
     res*=i;
}
```

This function requires no actual writes to the stack/physical memory as there are not any function calls / jumps. As this function only involves a single multiply, it qualifies as a “short loop” on the MSC814x and MSC815x devices, whereby the loop is entirely handled in hardware. Thanks to this feature, there are no change of flow penalties, no loop overhead either, so this effectively acts like a completely unrolled loop of multiplies (minus the memory cost).

Compared to the recursive routine, using the loop for 100 factorial saves approximately:

- 100 change of flows (pipeline cycle penalties)
- 100+ pushes to the stack (100x memory accesses)

For the above example, avoiding recursion savings can be estimated as follows:

The loop method’s change of flow savings from avoiding pipeline disruptions depend on the pipeline length and if there is any branch prediction available in the core hardware. In the case of the MSC8156’s 12 stage pipeline, refilling it would potentially be a 12 cycle penalty. As branch target prediction is available on the MSC8156, this may reduce some of this penalty significantly, but not completely. We can multiply the

estimated stall penalty by the factorial (iteration), which will indicate the additional active core clock cycles and thus active power consumption from the DSP core due to recursion.

The cost of recursion causing 100+ individual stack accesses is great, as even in internal device memory there is potentially an initial delay penalty for initial access. As these stack accesses will not be pipelined, initial memory delay is multiplied by the number of recursive calls. If we assume stack is stored low latency internal memory running at the core speed, initial latency still could be seen on the order of anywhere from 8-20 cycles. A 10 cycle latency for initial access would not be a problem if subsequent accesses were pipelined, meaning 100 reads have a total core stall time of 10 cycles, but in the case of recursion, we have non-pipelined accesses and thus 10 x 100 stalls, or 1000 additional core cycles of active clock consuming power.

In the above example, removing recursion and moving to loops reduces the total energy (power over time) consumed by the processor to complete the factorial function to less than half.

Reducing Accuracy

Rob Oshana brings up an interesting point in the article “Software Programming Techniques for Embedded DSP Software” [11] noting that often programmers will over-calculate mathematical functions, using too much accuracy (too much precision),which can lead to more complicated programs requiring using more functional units and more cycles.

In the case that 16-bit integers could be used as the signal processing application is able to tolerate more noise, but 32-bit integers are used instead, this could cause

additional cycles for just a basic multiply. A 16-bit by 16-bit multiply can be completed in 1 cycle on most DSP architectures, but a 32-bit by 32-bit may require more. Such is the case for the SC3400 DSP core, this requires two cycles instead of one, so the programmer is doubling the cycle time for the operation needlessly (inefficient processing and additional clock cycles where the core is consuming active dynamic power).

Low-power code sequences and data patterns:

Another suggestion from Oshana's article [11] is to look at the specific instructions used for an operation or algorithm. The programmer may be able to perform the exact same function with different commands while saving power, though the analysis and work to do this is very time consuming and detail oriented.

Different instructions activate different functional units, and thus different power requirements. To accurately use this, it requires the programmer to profile equivalent instructions to understand the power tradeoffs.

Obvious examples could be using a MAC when only the multiply functionality is needed. Less obvious comparisons, such as the power consumption between using a subtraction to clear a register versus the actual clear instruction require the programmer to profile power consumption for each instruction, as we may not know internally how the hardware goes about clearing a register.

Fixed Point vs Floating Point:

Another topic in regards to how to implement/optimize code at the algorithmic level involves the data type used, specifically if the algorithm or code is implemented in

fixed or floating point. As a rule, fixed point hardware, and thus the fixed point DSP, is cheaper than floating point hardware, but fixed point is more restrictive on the programmer. Of the devices discussed in this paper, the MSC815x series DSPs have fixed point DSP cores, and some devices in the C6000 family are fixed point, whereas other C6000 devices support floating point or a combination of both. If a programmer attempts to compile floating point code for a fixed point processor, they will probably experience a few challenges, as running ("emulating") floating point code on a fixed point DSP will result in significant amounts of overhead in cycle time, and thus power consumption.

Fixed point DSPs handle data in integer format, where these integers may be used to represent data with a fixed decimal point location and amount of precision for a number. All mathematical operations on a set of numbers assume the same scaling factor. This is opposed to floating point operations where, as the name implies, the decimal point of the number in question may be different from number to number within mathematical operations, and a much wider range of numbers may be represented using data representation in scientific notation (exponential format).

As some applications require greater precision and/or a wider range of numbers, these will generally be intended for use in floating point DSPs, and when moved to fixed point DSPs, will experience losses in both cycle efficiency and precision. A good comparison of the fixed point vs floating point application example need is expressed in [12], where the video and audio applications are compared. Video applications are expressed in terms of pixels, which are represented with relatively lower precision (less than 16 bits), as opposed to audio, where 32-bit by 32-bit multiplies are not uncommon. The need for higher precision in audio, according to [12], is due to the ability of the human ear to detect sound more effectively than the eye can process precision in moving

images. As such, audio applications are generally more geared towards floating point algorithms and processors; where as video applications can take advantage of fixed point processors.

When precision and number range are not an issue, and the programmer simply needs a way to run floating point code on a fixed point processor, updating the floating point algorithm to fixed point will result code that is far more power and cycle efficient than relying on floating point emulation in a fixed point processor. There are some software tools that enable a programmer to transform floating point code into fixed point at the cost of some precision in order to enable the programmer to take advantage of the cost of fixed point processors without the power and cycle tradeoffs [13].

FURTHER READING

This paper is aimed to be a self contained complete reference to power optimization in software, but there are some areas of software optimization currently unimplemented in commercial embedded programming, which may be of interest to the reader.

One main area of new research for software power optimization is in power aware compiler tools. As one might imagine, a compiler that automatically optimizes for power consumption will greatly speed the process laid out in this paper, thus reducing lead-time. One such optimizer is researched and discussed in [15]. While this discusses the idea of power focused software optimization, it focuses on a single DSP optimization technique, the use of multiple moves for single instructions (as was discussed in section 2.2 of this paper), while not covering other techniques. Despite this, maximizing just one power optimization technique will save the programmer significant time when considering large scale programs.

SOFTWARE SIMULATORS FOR POWER ESTIMATION

Another area related to power aware optimization and compilation tools is power estimation tools for software. Hardware may not always be available (due to limited resources, early in the design cycle, etc). In this case, the user may want to consider the availability of software simulators that provide power analysis information. Jacome and Russel investigated this for the embedded arena in [16]. They investigated power estimation for an embedded microcontroller (non-DSP) and focused on frequency, data path, control path, and total execution time with very high accuracy (within 8% in their testing). The reader should note that as the tool discussed in this paper is for a general

microcontroller, results will be slightly different for a high performance DSP, which will have some dependency on the type of instruction used, as discussed earlier in this paper.

There are also tools available in the open source market such as the SimpleScalar simulator tool using the Wattch plugin [17], which allow for C code power profiling. Wattch was designed to work with SimpleScalar and was tested against the Intel Pentium Pro and Alpha 21264 and noted as having within 10% accuracy in power estimates.

In the case of Freescale's StarCore architecture, JouleQuest [18] was developed for the SC140e DSP core (which the SC3400 and SC3850 cores are based off of). With this tool, as reported in JouleQuest's article, power modeling for the DSP core and caches/local memory is within a 5-10% error margin. As such, tools like JouleQuest tool can be quite useful to core DSP algorithm designers in order to help power optimization in the stead of having available hardware for power profiling.

The potential problem with the above software tools is accuracy and reliability. While a DSP processor core (such as the StarCore SC1400) may be present in the MSC8144, it is present in many other processors as well (MSC71xx, MSC8101, MSC8103, MSC8122, and Motorola wireless cellular phones), all of these different implementations will have different power consumption values due to different process technology, different processor architecture, different bus and memory connectivity. Because of this, significant time is required for a tool like JouleQuest to be useful in a different implementation from the original hardware the tool is based on.

FUNCTIONAL UNIT CLOCK GATING:

While not available in the MSC815x or C6000 DSP processors, functional unit clock gating – gating the clock tree to certain functional units within the processor's core

– is another method available for the software engineer to effectively reduce system power.

As previously noted in this paper, the power consumed from clocking in a processor is significant. In the case of the MSC8156's SC3850 core, for example, while executing control code that may only make use of one AGU and one DALU per cycle, if we could gate the clock to the 2nd AGU and the 2nd, 3rd, and 4th ALUs, we could save a significant amount of power. For reference, if we look back to the power saving the MSC8156 had when being put in WAIT and STOP state over IDLE state in our MJPEG example, we may be able to gather some idea.

This technology has been tested in FPGA solutions for experimentation purposes [19] and implemented in order to design a low power ALU which is broken up into different functional units so that part of the ALU can be gated based on instruction opcode. Using this architecture, if an operation being executed is an arithmetic operation, the unused logic portion of the ALU will be clock gated.

In addition to clock gating by functional unit per instruction, functional unit clock gating can be used to gate the clock to different functional units as they are seen/issued in the processor core's pipeline: pre-fetch, fetch, instruction decode, dispatch, etc. Many processor cores, including Freescale's and TI's solutions, include Branch Prediction, which uses certain algorithms to predict if a branch in code will or will not be taken, and then pre-fetch the instructions from the where the BTB logic expects the code to jump to. One group [20] explored taking advantage of the stalls and wasted power in the pipeline stages when, for example, the BTB makes an incorrect prediction by gating the clock to the affected functional units to save as much as 13.93% in processor's power consumption.

This paper doesn't just look at clock gating during incorrect BTB prediction, but also gating unused pipeline stages, gating affected functional units during pipeline stalls, and gating unused logic similarly to our discussion about control code on the MSC8156. Although this paper is focused more at the SOC hardware architecture level, it gives us insight into the possible savings available when taking advantage of such a feature. The disadvantages noted for using such a feature would be architecture dependant, but there is some noted risk of cycle overhead noted in the paper. As with the STOP and WAIT modes in the MSC815x, based on the above research, it would be sensible to consider this feature on applicable architectures in appropriate scenarios (such as in control code – where certain functional units will remain idle for notable periods of time).

Despite the fallbacks of the topics for further reading, the strengths that they can all be used in combination with the techniques covered in this paper make them potentially quite useful once research and development has matured.

SUMMARY AND CLOSING REMARKS

In order to provide the reader with tools to optimize software for power, over thirty different optimization techniques in the areas of low power modes, current and voltage controls, memory optimization, data path optimization, and algorithmic strategies have been discussed. A summary of those techniques are provided in the table below:

Category	Technique	Impact
Hardware Support	Power Gating: Via a VRM or processor supported interface, switch off current to specific logic or peripherals of the device.	HIGH
Hardware Support	Clock Gating: Often provided as device low power modes, maximize the amount of clocks that can be shut down for an application	HIGH
Hardware Support	Voltage and Clock scaling: where available, reduce frequency and voltage.	Processor dependent
Hardware Support	Peripheral Low Power modes: gating power/clock to peripherals.	Medium-High
Data Flow	DDR Optimizing Timing: Increasing timing between ACTIVATE commands	Low
Data Flow	DDR Interleaving: used to reduce PRECHARGE/ACTIVATE combinations	HIGH
Data Flow	DDR optimization of software organization: organizing buffers to fit into logical banks to avoid PRECHARGE/ACTIVATE commands	Medium
Data Flow	DDR General Configuration: avoid using modes such as open/closed page mode, which would force a PRECHARGE/ACTIVATE after each write	HIGH
Data Flow	DDR Burst Accesses: organize memory to make full use of the DDR burst size. This includes alignment and data packing.	Medium
Data Flow	Code Size: Optimize code and data for minimal size via compiler tools	Application dependent
Data Flow	Code Size: code packing	Medium
Data Flow	Code Size: creating functions for common tasks	Application dependent
Data Flow	Code Size: Utilized combined function instructions (multiple instructions in one, which save size and cycles)	Processor dependent
Data Flow	Code Size: Use tools for compression on the fly	Processor dependent
Data Flow	Parallelize and pipeline accesses to memory	Medium
Data Flow	Use constants and avoid zeroing out memory	Processor dependent
Data Flow	Cache: Layout memory to take advantage of cache set associativity	Application dependent
Data Flow	Cache: Use write-back model when available and feasible for application	Application dependent
Data Flow	Cache: use prefetching to bring data in ahead of time and avoid miss penalties and extra dead clock cycles	Application dependent
Data Flow	Cache: array merging	Application dependent

Table 1: Summary of Power Optimization Techniques

Data Flow	Cache: interchanging	Application dependent
Data Flow	Take advantage of DMA for memory movement	Medium
Data Flow	Coprocessors: Use to perform functions instead of core	Medium
Data Flow	System Bus Configuration: configure bus to minimize stalls and bottlenecks	Application dependent
Data Flow	Peripheral speed grades and bus width: optimize per usage needs.	Application dependent
Data Flow	Peripheral to core flow: use interrupt processing when possible	HIGH
Algorithmic	Compiler Optimization levels: use compiler optimization tools to optimize for performance to minimize cycle time in critical areas, and optimize for code size elsewhere	Medium
Algorithmic	Instruction Packing: maximize code to functionality efficiency	Medium
Algorithmic	Loop Unrolling: maximizes parallelism, minimizes active clock time	HIGH
Algorithmic	Software Pipelining: another method to maximize parallelism and minimize active clock time	HIGH
Algorithmic	Eliminating Recursion: save cycle time from function call overhead	HIGH
Algorithmic	Reducing accuracy: saving cycles by reducing calculations	Application dependent
Algorithmic	Low power code sequences: using equivalent functions via a lower power set of instructions	Processor dependent

Table 1: Summary of Power Optimization Techniques, cont.

In the process of explaining how to optimize software to minimize power consumption we have also covered some basic details about the DSP devices from Freescale and Texas Instruments, provided background into how low power modes work, how DDR and caches work, and how the compiler works in order to assist the programmer.

Numbers and references were provided for many of these optimization techniques as they apply to the hardware tested in this work, but the reader must understand that as every application is different, and will work differently on other hardware, that power consumption must be profiled as discussed in Section 2.

BIBLIOGRAPHY

[1] Robert Oshana, "*DSP Software Development Techniques for Embedded and Real-Time Systems*". (Maryland Heights: Newnes, 2005).

[2] Wikipedia, "Electron Mobility." Last modified on August 21, 2011. http://en.wikipedia.org/wiki/Electron_mobility.

[3] Allegro MicroSystems, Inc. "Fully Integrated, Hall Effect-Based Linear Current Sensor with Voltage Isolation and a Low-Resistance Current Conductor," Revision 6, October 2006, www.allegromicro.com/en/Products/Part_Numbers/0704/0704-015.pdf

[4] Ping Yeung and Erich Marschner, "Power Aware Verification of ARM-Based Designs," EE Times, November 3, 2010. http://www.eetimes.com/design/embedded/4210422/Power-Aware-Verification-of-ARM-Based-Designs-.

[5] Micron Technology, Inc. "Technical Note TN-41-01: Calculating Memory System Power for DDR3," Revision B. August 2007.

[6] Freescale Semiconductor, "MSC8156 Reference Manual," Revision 2, June 2011.

[7] Kojima et al., "Power analysis of a programmable DSP for architecture/program optimization," Proc. IEEE Int. Sym. On Low Power Electronics, pp. 26-27, Oct. 9-11, 1995.

[8] Dhireesha Kudithipudi, “Caches for Multimedia Workloads: Power and Energy Tradeoffs,” IEEE Transactions on Multimedia, Vol. 10, No. 6, October 2008.

[9] Ibrahim et al., “Performance and Power Consumption Trade-offs for a VLIW DSP,” Proc. Proceedings of the International Symposium on Signals, Circuits and Systems, pages 197 - 200, Isia, Romaina, July, 2009.

[10] Wayne Wolf, “Basics of Programming embedded processors part 8,” September 11, 2007, http://www.eetimes.com/design/embedded/4007176/The-basics-of-programming-embedded-processors-Part-8.

[11] Robert Oshana, “Software Programming Techniques for Embedded DSP Software,” April 19th, 2007, http://www.dsp-fpga.com/articles/id/?2548.

[12] Gene Frantz, “Comparing Fixed and Floating Point DSPs.”, spry061, 2004. http://www.ti.com/lit/wp/spry061/spry061.pdf.

[13] A. Cilio and H. Corporaal, “Floating Point to Fixed Point Conversion of C Code.” Lecture Notes in Computer Science - Compiler Construction, Springer Berlin / HeidelbergIsbn. Volume: 1575. 1999.

[14] David A. Ortiz and Nayda G. Santi, “Impact of Source Code Optimizations on Power Consumption of Embedded Systems,” Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008:133 – 136.

[15] Lorenz et al., “Compiler based Exploration of DSP Energy Savings by SIMD Operations,” Asia and South Pacific Design Automation Conference, pp. 838-841, (ASP-DAC'04), 2004.

[16] Jeffry T. Russell and Margarida F. Jacome, “Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors,” Proceedings

of the International Conference on Computer Design (ICCD '98). IEEE Computer Society, Washington, DC, USA: 328.

[17] Brooks, et al. “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations,” SIGARCH Comput. Archit. News 28, 2 (May 2000), pp. 83-94,

[18] Mathur et al., “JouleQuest: An Accurate Power Model for the StarCore DSP Platform,” 20th International Conference on VLSI Design (VLSI’07). pp. 521-526, IEEE Computer Society, 2007.

[19] M. Kamaraju, et al, “Power Optimized ALU for Efficient Datapath,” International Journal of Computer Applications (0975-8887). Vol 11-No.11, December 2010.

[20] N. Mohyuddin, K. Patel, and M. Pedram, "Deterministic Clock Gating to Eliminate Wasteful Activity Due to Wrong-Path Instructions in Out-of-Order Superscalar Processors," Proc. of Int'l Conference on Computer Design, pp. 166-172, Oct. 2009.

[21] James Morris, “65nm to 45nm: Process Technology Explained,” TechRadar, November 7th 2007, http://www.techradar.com/news/world-of-tech/future-tech/65nm-to-45nm-process-technology-explained-147819.

[22] Mike Tien-Chien Lee and Vivek Tiwari, “A Memory Allocation Technique for Low-Energy Embedded DSP Software,” IEEE Symposium on Low Power Electronics, October 9-11, 1995.

[23] Jim Patterson and John Dixon, “Optimizing Power Consumption in DSP Designs,” October 25, 2006, http://www.ti.com/lit/wp/spry089/spry089.pdf.